Runtime MPI Correctness Checking with a Scalable Tools Infrastructure

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Diplom-Informatiker Tobias Hilbrich
geboren am 5. Oktober 1983 in Dresden

Gutachter: Prof. Dr. rer. nat. Wolfgang E. Nagel, Technische Universität Dresden
Prof. Dr. rer. nat. habil. Thomas Ludwig, Universität Hamburg

Tag der Einreichung: 18. Dezember 2014
Tag der Verteidigung: 08. Juni 2015

Whether or not what you do has the effect you want,
it will have three at least you never expected,
and one of those usually unpleasent.
–From Robert Jordan's novel "The Path of Daggers"

Abstract

Increasing computational demand of simulations motivates the use of parallel computing systems. At the same time, this parallelism poses challenges to application developers. The Message Passing Interface (MPI) is a de-facto standard for distributed memory programming in high performance computing. However, its use also enables complex parallel programing errors such as races, communication errors, and deadlocks. Automatic tools can assist application developers in the detection and removal of such errors. This thesis considers tools that detect such errors during an application run and advances them towards a combination of both precise checks (neither false positives nor false negatives) and scalability. This includes novel hierarchical checks that provide scalability, as well as a formal basis for a distributed deadlock detection approach.

At the same time, the development of parallel runtime tools is challenging and time consuming, especially if scalability and portability are key design goals. Current tool development projects often create similar tool components, while component reuse remains low. To provide a perspective towards more efficient tool development, which simplifies scalable implementations, component reuse, and tool integration, this thesis proposes an abstraction for a parallel tools infrastructure along with a prototype implementation. This abstraction overcomes the use of multiple interfaces for different types of tool functionality, which limit flexible component reuse. Thus, this thesis advances runtime error detection tools and uses their redesign and their increased scalability requirements to apply and evaluate a novel tool infrastructure abstraction. The new abstraction ultimately allows developers to focus on their tool functionality, rather than on developing or integrating common tool components. The use of such an abstraction in wide ranges of parallel runtime tool development projects could greatly increase component reuse. Thus, decreasing tool development time and cost. An application study with up to $16{,}384$ application processes demonstrates the applicability of both the proposed runtime correctness concepts and of the proposed tools infrastructure.

Contents

1 Introduction

The scientific community places more faith in computation than is justified. [85]

Ensuring correctness of applications is an important task during the development of parallel applications. This thesis advances tools that aid application developers in the removal of program defects during this task, as well as new concepts and methods for parallel tool infrastructures that simplify the development, maintenance, and improvement of such tools. A multitude of techniques, best practices, and tools can aid application developers in locating and removing software defects, this thesis focuses on runtime correctness tools for a message passing paradigm that targets high performance computing systems.

1.1 Motivation

The development of massively parallel software is crucial to satisfy increasing computational demands of simulations. At the same time, the High Performance Computing (HPC) systems that support such parallelism follow hardware trends towards increasing compute core counts. As an example, the November 2013 issue of the Top500 list [152] includes top five system such as the K computer at the RIKEN Advanced Institute for Computational Science with a total of 705,024 cores and the Sequoia system at the Lawrence Livermore National Laboratories with a total of 1,572,864 cores. Many parallel applications utilize these systems for a wide range of application, including weather prediction [92], cosmology [61], medicine [127], and quantum chromodynamics [37]. Parallel programming abstractions in the form of libraries, runtimes, or language extensions enable such simulations. At the same time, parallel-programming paradigms introduce additional sources for software defects. Program state that is distributed across multiple threads or processes complicates the removal of such defects. Thus, the development of parallel software is challenging and programming errors occur [15, 60, 65, 80, 95, 123, 142]. Also the use of parallel debuggers such as Totalview [128] and Allinea DDT [5] during application development [79] suggest the presence of faults in parallel applications. Finally, studies [26, 109] also find that tools for debugging and fault detection facilitate efficient development workflows.

The Message Passing Interface (MPI) [114] is a de-facto standard for distributed memory programming [23, 33, 79]. MPI enables the development of highly scalable applications [37, 61, 92, 127], but offers few extensions to enforce its correct use. The standard includes more than one hundred functions to which specific rules and restrictions apply. Particularly, software faults that involve incorrect MPI usage can manifest into failures that corrupt application results, cause an application crash, or as a deadlock. In addition, some defects may stay undetected and can silently corrupt the results of an application, while other defects only infect the application state for particular executions (non-determinism), at specific scales, or on specific compute systems.

Tools can aid application developers in the removal of MPI related software defects with a variety of techniques [57]. Automated runtime correctness tools form one of these techniques and detect usage errors of the MPI standard during an application run. Application developers can use such tools to understand and analyze failures (defects that caused abnormal application behavior), as well as several defects that only cause failures for specific MPI implementations or executions. This thesis compares existing runtime correctness tools for MPI and reveals that current approaches either lack precision or scalability. This thesis defines approaches of limited *precision* as approaches that purposefully use methods that can yield false positives or false negatives, i.e. methods that can fail to detect failures under some conditions or report failures for correct applications.

When MPI related failures occur at a scale that renders existing runtime correctness tools impractical, application developers are forced to reduce the size of their test case in order to decrease scale. This in

turn may impact the control flow of the application, as well as it may impact non-deterministic choices during the execution. As a result, some failures may not occur in smaller configurations and need to be understood at a certain scale [7, 51, 101, 148, 160, 161]. Runtime correctness tools that do not scale sufficiently become impractical then. This thesis designs techniques and algorithms that provide runtime error detection capabilities at an increased scale, without limiting their precision.

The development of runtime correctness tools requires components for common tasks, e.g., instrumentation, communication, and tool startup/shutdown. Since the development of portable, efficient, and scalable tool components is challenging [104, 130], reusable components preserve lessons learned and simplify tool development. Parallel tool infrastructures or frameworks—*tool infrastructures* in the following—can both directly provide tool components, e.g., an instrumentation service, and can provide abstractions that allow a reuse of tool functionality. Thus, tools that use the same tool infrastructure can share parts of their implementations. An evaluation of existing tool infrastructures in this thesis reveals that none of the studied infrastructures provides all the components that MPI runtime error detection tools require. Further, abstractions with tree networks—which provide a simple and widely used scalability concept—use distinct interfaces for specific sets of tool components. Thus, components written for one of these interfaces will be incompatible for use with the other interfaces, e.g., a piece of tool functionality that usually operates on the application processes cannot be migrated to a higher hierarchy layer of the tree directly. This unnecessarily restricts component reuse.

Thus, this thesis follows two goals: First, it advances runtime correctness tools for MPI with concepts and algorithms for scalable and precise checks. Second, it proposes a novel abstraction for a parallel tools infrastructure that simplifies the development of such a correctness tool. The abstraction and the assumptions for this infrastructure must apply to the correctness tool use case, but should be general enough to apply to a wider range of runtime tools. This approach allows a study of how tool infrastructures can support additional requirements for runtime correctness tools and it evaluates how advanced infrastructure features can facilitate tool development. Such capabilities should particularly include an abstraction that enables reuse of parts of the tool implementation across different tools. In an ideal situation, an infrastructure for parallel tools should not only provide a set of features that suffices to implement a tool, but should allow tool developers to focus on the implementation of their actual tool analyses, rather than distracting them with the integration, use, or extension of infrastructure components. Challenging correctness analyses such as deadlock detection then provide a means to test both the scalability features and the abstraction of the infrastructure that this thesis proposes. At the same time, these correctness capabilities advance the scalability of previous precise approaches for runtime MPI correctness tools.

1.2 Contributions

This thesis proposes a novel abstraction for parallel tool infrastructures, along with a prototype implementation called GTI (Generic Tools Infrastructure). The abstraction combines an instrumentation service with concepts that organize tool functionality into reusable *modules*. Existing infrastructures also use module concepts, but apply restrictions that limit their composability and reuse. A key contribution of this new abstraction is that modules can specify their own use case dependent interfaces. An extension of event-action mappings [169] then allows tool developers to flexibly associate the interfaces of the modules with events of interest. A notion of hierarchies for tool owned processes (or threads) provides a means to offload and distribute the processing of a tool. Formal definitions then describe how the mappings—of module interfaces to events—interact with the hierarchies of tool processes/threads. This includes an integration of an event aggregation concept to enable tool scalability.

Thus, the proposed abstraction both relies on existing tool infrastructure concepts that demonstrated applicability and scalability, as well as a novel mapping-based concept. The mappings employ a specification language to describe a tool. A tool instantiation workflow then uses a given tool specification to connect components such that they form the overall tool. This includes components that the parallel tools infrastructure provides, use-case-specific instrumentation components, and user-provided modules. Ultimately, the proposed abstraction allows tool developers to provide all of their tool specific analyses

in the form of modules, while previous infrastructures required developers to provide some tool components themselves, e.g., instrumentation. Further, this thesis provides novel techniques that overcome limitations of previous infrastructures or tool components. This includes an algorithm for order preserving event aggregation, a scheme to handle application crashes, and a communication system that allows direct communication between tool processes of the same hierarchy level.

Based on the GTI prototype, this thesis designs a prototype for an MPI runtime correctness tool called MUST (Marmot Umpire Scalable Tool). MUST combines the correctness checks of two existing tools and provides precise checks for challenging correctness analyses such as deadlock detection. Novel distributed schemes in this thesis allow these correctness analyses to employ the scalability services of the proposed tools infrastructure abstraction. Particularly, this includes hierarchical checks for MPI collective operations and a transition system that explicitly follows matching decisions of the MPI implementation to derive a distributed deadlock detection. An analysis of the theoretic time complexities of the proposed correctness analyses underlines a high degree of scalability. Overall, the MUST prototype can completely rely on the services of the GTI prototype.

Related tool development projects demonstrate that the GTI prototype is applicable to other types of runtime tools as well. Additionally, application studies with synthetic stress tests and two benchmark suites demonstrate the applicability of the GTI and MUST prototypes on two different compute architectures. The synthetic stress tests evaluate the scalability of both the proposed tool infrastructure abstraction and of the distributed correctness analyses under a weak scaling scenario. The strong scaling benchmarks then demonstrate tool overheads for benchmarks that are derived from real world applications. The experiments use up to $4{,}096$ application processes on a Linux cluster at the Lawrence Livermore National Laboratory and up to $16{,}384$ application processes on a BlueGene/Q system at the Forschungszentrum Jülich. The synthetic stress tests highlight the scalability of the proposed correctness analyses for different MPI communication kernels. The distributed deadlock analysis of MUST can handle both of the widely used HPC benchmarks at $2{,}048$ and $16{,}384$ processes respectively. It yields low to moderate overheads for thirteen out of nineteen kernels. Additionally, a comparison with a state-of-the-art precise deadlock detection approach highlights performance improvements of two orders of magnitude for 512 processes already.

1.3 Thesis Organization

Chapter 2 introduces types of MPI usage errors and existing approaches that target their automatic detection. This thesis focuses on approaches that operate during the runtime of an application. In order to assess the features of these approaches and their scalability, the chapter first provides a tool architecture classification and secondly a set of metrics. These metrics and the architecture types highlight that both approaches with a high potential for scalability and approaches with wide ranges of precise checks (no false positives or false negatives) exist. However, no current runtime correctness approach provides both at the same time. This deficit motivates an extension towards a precise approach that retains the ability to scale with an application.

An introduction to common tool components for runtime MPI correctness tools motivates the use of parallel tool infrastructures in Chapter 3. Afterwards, a requirements list summarizes key tool infrastructure features for runtime correctness tools. The chapter introduces scalability features and abstractions of existing tool infrastructures to compare them to the former requirements.

Chapter 4 introduces a new abstraction for a parallel tools infrastructure and its prototype implementation GTI. A terminology introduction presents key terms and concepts. Afterwards, formal specifications describe the ideas behind this abstraction in detail. A description of advanced concepts then provides the details for the proposed scalability and crash-handling services. Algorithms for an instantiation workflow connect and elaborate how a prototype implementation can implement the abstraction. Finally, a novel algorithm for order preserving event aggregation in a tool hierarchy forms a basis for an important scalability feature.

Based on the GTI prototype, Chapter 5 introduces the prototype of the new runtime MPI correctness

tool MUST, along with its distributed correctness analyses. A description of the tool design highlights its key components and structure. The chapter then details three correctness analyses that impose challenges for scalability in detail: Point-to-point handling, collective handling, and deadlock detection. The analysis of MPI point-to-point operations introduces a concept to detect MPI datatype matching defects in a distributed manner, without sacrificing precision. Afterwards, the chapter introduces a hierarchical correctness analysis for collective operations that also combines precision with scalability. Finally, an overview of state-of-the-art MPI deadlock detection algorithms motivates a combined approach that employs a transition system.

Chapter 6 uses synthetic stress tests and two benchmark suites to evaluate the applicability and scalability of the prototype implementations MUST and GTI. The synthetic stress tests compare GTI and MUST to an analysis of their theoretic time complexities. Additionally, a comparison to a centralized reference implementation highlights how previous precise runtime correctness tools would react for such scenarios. Strong scaling benchmarks then introduce more complex MPI usage scenarios. Chapter 7 concludes this thesis and summarizes ongoing and future work.

2 State-of-the-Art in MPI Verification

> *Despite MPD's [a process manager] small size and apparent simplicity, errors have impeded progress toward code in which we have complete confidence. Such a situation motivates us to explore program verification techniques.* [111]

This chapter starts with a short terminology introduction (Section 2.1) and highlights debugging challenges for parallel applications. An introduction to the distributed memory programming concepts of MPI then details the environment under consideration (Section 2.2). Examples of MPI usage errors illustrate the software defects that this thesis considers (Section 2.3). Afterwards, Section 2.4 introduces approaches that aid application developers in the detection and removal of MPI related software defects. Runtime tools that automatically detect defects that involve incorrect usage of the MPI standard are the focus of this survey (Section 2.5). Since this thesis considers tool scalability as a requirement for tool applicability, Section 2.6 categorizes the tool architectures of such runtime tools. Finally, a comparison of the individual runtime approaches motivates the specific improvements that this thesis addresses (Section 2.7).

2.1 Terminology and Parallelism

This thesis considers techniques that aid application developers in the removal of software defects; or commonly *bugs*. This document uses the term *defect* to refer to a problem in the software, which violates the specification of the software or of another component that the software uses. Following this terminology [82, 173], defects cause *infections* that are incorrect application states. Finally, infections can cause *failures* that are visible derivations from the software specification, e.g., an obviously wrong result, an application crash, or a deadlock.

Defects occur in sequential as well as parallel applications. However, parallelism gives rise to additional types of defects. First, an infection on one application process or thread can pass to further processes or threads. As a result, failures may occur on processes or threads that have no defect. In addition, parallel applications give rise to non-deterministic failures that only occur in some executions of an application. Differences in thread or process schedules or in the behavior of used software components enable this non-determinism. Data races between threads are a primary example; consider two threads that both use a shared variable x. A software specification could require that the first thread reads the current value of x and the second thread assigns a new value to x only afterwards. If the threads lack the necessary synchronization to enforce such an order, then some executions can use the correct order, while others can result in a failure or an infected state. The problem of non-determinism especially challenges debugging workflows since they usually depend on reliably reproducing a failure.

This thesis focuses on defects that occur in parallel applications that use the Message Passing Interface (MPI) (The following section introduces this interface). The interface uses a process abstraction that does not suffer from classical data races, but offers other sources for non-determinism. Programming techniques like code inspection [46] and defensive programming [30] offer techniques to avoid MPI usage related software defects in the first place. Inspection techniques aid in an early removal of defects and directly operate on the source code. Defensive programming defines good practices that should be adhered to, e.g., assertion and return value checks, and dangerous practices that should be avoided, e.g., non-deterministic MPI constructs. Parallel debuggers such as DDT [5], Totalview [128], and an extension of Ladebug [12] provide debugging environments that simplify the identification of defects based on observed failures. State inspection tools like STAT [7] augment these observation techniques for use cases that involve hundreds of thousands of application processes.

This thesis focuses on automatic verification approaches that address failures that involve the use of MPI. Such automatic verification techniques use the MPI standard documents as a specification and detect violating MPI usage with different methodologies (Section 2.4). These techniques report violations to restrictions of the MPI standard as *failures*. As opposed to the previous definition of failure, a reported violation may not cause a failures in some cases. This document uses the term failure for instances of detected MPI restrictions, irrespective whether they are visible in the execution of an application. This notion also highlights the limitations of automatic verification approaches. As long as they only use the MPI standard as their specification, they cannot detect all failures. As an example, automatic verification can reveal an incorrect communication between two processes, but it cannot reveal whether a correct communication fails to transfer the right data.

The failures that automatic verification approaches detect result from defects in the software or of one of its components. A strength of automatic verification is that it can specifically attempt to investigate different process schedules in order to reproducibly detect non-deterministic failures. Depending on the approach, automatic verification can pinpoint developers to the actual defects, hint at them, or only provide information on the failure itself. As a result, automatic MPI usage error detection approaches are synergetic with programming techniques and parallel debugging techniques. Tool integrations [98] of automatic MPI verification tools with parallel debuggers highlight this.

2.2 The Message Passing Interface

MPI evolved as a standardized library for message passing on distributed memory HPC systems. It replaces predecessor approaches that existed in competition or where only available for some compute systems. The interface uses processes to utilize parallel computing elements, such as compute cores on a CPU. All processes execute the same application source code and can only be differentiated by their *rank*, which for p processes is an identifier between 0 and $p-1$. The following uses the symbol p to represent an application process count. In order to cooperate, processes then call functions of MPI to communicate with each other. This thesis describes techniques and algorithms to detect MPI usage errors at runtime. These techniques can also apply to other message passing paradigms such as MCAPI [38, 151].

2.2.1 Communication Operations

This document uses the terms of the MPI standard except for the term *operation* that refers to an invocation of an MPI function here. A primary intent of the MPI functions is the specification of data transfers between processes. Subsequently, the MPI operations that enable these transfers are also a primary target for automatic verification approaches. While the MPI standard documents introduce the individual operations, their restrictions, and semantics in detail, the following list briefly introduces groups of communication operations to which this thesis refers:

point-to-point: Involve a sender and a receiver process where the sender transfers data to the receiver, e.g., `MPI_Send` as a send operation and `MPI_Recv` as a receive operation; These operations return the control flow to the calling application process only when they completed, unless the send transfer uses a buffering semantics;

Nonblocking point-to-point: Initiates send or receive operations without blocking the control flow of the calling application process, a *request handle* identifies the initiated send or receive transfer, e.g., `MPI_Isend` initiates a nonblocking send;

Completion: Can complete one or multiple send/receive transfers that are associated with a set of requests; *Wait* completions (e.g., `MPI_Wait`) block the control flow until such a transfer completes and *test* completions (e.g., `MPI_Test`) return the control flow after a finite amount of time; and

Collective: Transfer data between a group of processes that is identified with a *communicator* handle and can block the control flow until the operation completes.

The handles in these operations identify MPI objects such as an outstanding communication or a process group. This document uses the term *resource* to refer to these objects. Further groups of operations include initialization and management functions, one-sided communication operations, and operations for parallel I/O. Various restrictions in the MPI standard apply to the operations of the initialization and management group. These restrictions often only involve a single process, and thus are less challenging for efficient automatic verification techniques than the communication operations that often involve at least two processes. As a result, this thesis focuses on the communication functions, while actual automatic verification approaches must also handle these management functions. The respective handling also exists in the prototype that later chapters in this thesis describe. The other operation types (one-sided and parallel I/O) represent orthogonal extensions of the original MPI specification and motivate future extensions of this thesis.

2.2.2 Non-Determinism

MPI provides multiple operations that can introduce non-determinism. Receive transfers that accept a send from any process—with the source argument `MPI_ANY_SOURCE`—are an example for such operations. If two or more send transfers exist that can satisfy such a receive operation, then the behavior of an application depends on the matching decisions of the MPI implementation. Matching decisions of the MPI implementation, in turn, depend on the timing of the individual operations. If both sends occur *at about the same time* the MPI implementation can select either send operation for the receive. However, if one send precedes the other by a specific time threshold (a factor that depends on the latency between the involved processes), then the implementation will select the first send. Such non-determinism can introduce schedule dependent failures. The term *interleaving* refers to one potential schedule in the following. As an example, a receive of process 0 can match send operations of both processes 1 and 2. The MPI standard requires that send and receive operations specify matching MPI datatypes, which describe the payloads of the transfers. The MPI datatypes could match between the transfers of processes 0 and 1, while a mismatch could occur for processes 0 and 2. Thus, the defect of the potential MPI type mismatch manifests as an inconsistent state or a failure for interleavings where the second pair of transfers match. If the timing of the operations usually prefers the first match, then the failure may be very hard to reproduce.

Further MPI operations that can introduce non-determinism include wait and test operations that can complete subsets of communication requests. Additionally, the MPI standard includes implementational freedoms such as the processing order of the `MPI_Startall` operation, which initiates multiple non-blocking point-to-point operations.

2.2.3 Versions

A forum develops and improves MPI to include new features and corrections. While MPI-3.0 [114] was recently released, this thesis addresses MPI-1.3 [113] unless stated differently. MPI-1.3 includes the core services of the interface that suffice for many applications. The additional features of later versions primarily add smaller missing features to these core services, which the approaches in this thesis usually include, and orthogonal capabilities such as one-sided communication or parallel I/O operations. Additionally, MPI libraries are mostly backwards compatible, i.e., an MPI-3.0 implementation usually supports an MPI application developed for MPI-1.3, as well as tools for earlier MPI versions. Thus, an MPI runtime correctness tool for MPI-1.3 can often correctly operate with an MPI application that uses functionality of a newer version of the standard. However, the tool then only checks for a subset of the MPI related failures. Multi-threading support and a process creation/management interface are notable extensions of MPI-2.0 that can impact the correct operation of an automatic verification tool for MPI-1.3.

MPI-2.0 introduced support levels for threads. The first three support levels define that only one thread per process is active in an MPI call, while the highest level—`MPI_THREAD_MULTIPLE`—allows threads to concurrently issue MPI calls (with specific exceptions). The approaches and MPI examples in

Process 0	Process 1
MPI_Recv(from:1, count:1, type:MPI_INT)	MPI_Send(to:0, count:1, type:MPI_DOUBLE)

Figure 2.1: *A type signature mismatch for a pair of point-to-point operations as an example defect (MPI_INT≠MPI_DOUBLE).*

Process 0	Process 1
MPI_Bcast(root:0)	MPI_Barrier()

Figure 2.2: *If processes take different control flow choices they can arrive in a collective mismatch, as in this example defect.*

this thesis assume any of the first three levels. Support for MPI_THREAD_MULTIPLE requires a generalized deadlock handling scheme [73, 76] that includes information from the threading paradigm along with adaptions for other correctness analyses. The necessary incorporation of a threading scheme such as OpenMP [27] is beyond the scope of this thesis. An ongoing working group [36] in the MPI forum considers extensions to MPI that would simplify tool support for the highest thread level in upcoming versions of MPI.

The process creation and management interface of MPI-2.0 allows an MPI application to connect to or to start additional processes. Without extended support for this interface, tools can analyze the communication operations that occur in the original and/or the added process sets, i.e., retain a large amount of their functionality. Further support for communication between the process sets requires that tools can merge their state at runtime, which is an implementational challenge, but not a limitation to the methods in this thesis. Thus, such an extension remains a target for future work.

2.3 MPI Usage Errors

Existing studies of MPI usage errors [34, 97, 108] provide classifications and a taxonomy for MPI usage errors. This section introduces exemplary erroneous MPI usage scenarios that serve for differentiation and elaboration purposes in subsequent sections. Thus, these examples allow a distinction of specific tool approaches, rather than an overview over all types of usage errors, as in a classification or a taxonomy. For the taxonomy presented in one of these studies [34], all examples in this section either belong to the class of *synchronization* errors or the class of *mismatch* errors.

The examples in this section (Figures 2.1–2.8) use a shortened notation to represent incorrect MPI applications. The examples assume that MPI was initialized and no pending communication operations exist. Additionally, each operation only highlights key arguments. Especially the concept of MPI communicators provides a mechanism to subgroup MPI processes and to decouple communication operations from each other. All examples assume the use of the default communicator MPI_COMM_WORLD, which includes all processes. The operations in these examples include the collectives MPI_Bcast and MPI_Barrier; the point-to-point operations MPI_Send, MPI_Recv, and MPI_Isend (the latter is nonblocking); and the completion operation MPI_Wait. The example in Figure 2.8 additionally uses the MPI_Finalize operation to indicate when the application ends its MPI usage. The number of columns in Figures 2.1–2.8 indicates the number of MPI processes for each example, except for the scenario in Figure 2.5 that uses a variable number of processes.

Process 1 in Figure 2.1 uses a send operation to transfer data to a receive operation on process 0. However, both operations specify incompatible MPI datatypes to illustrate a type matching defect. MPI uses MPI datatypes to support heterogeneous platforms and requires that sending and receiving transfers specify datatypes that match. Such defects may remain undetected if all bytes that the send operation transfers fit into the receive buffer.

The example in Figure 2.2 erroneously executes distinct collectives on two processes. For each MPI communicator, only one collective operation may be active at a time. Thus, for a correct MPI program,

Process 0	Process 1
`MPI_Bcast`(root:0)	`MPI_Bcast`(root:1)

Figure 2.3: *Even if processes execute collective operations of the same type, they can specify incompatible arguments as in this example. For* `MPI_Bcast` *the MPI standard requires all participating processes to identify the same process with the* root *argument.*

both processes must execute collectives of an identical type with compatible arguments in the same order; i.e., first both processes execute the `MPI_Bcast` operation and then both processes execute the `MPI_Barrier` operation. Such a defect may manifest as a hang, as a crash (within the collective operation or at a later time), or remain undetected. The latter is possible since most MPI implementations use point-to-point operations to realize collective operations. The point-to-point operations will often be buffered—unless the MPI standard requires that synchronization takes place—if communication payloads are low.

The example in Figure 2.3 highlights a further usage error for a collective transfer. While both processes use a consistent operation type, they use inconsistent data (*root* argument) for their individual operations. The MPI standard requires that all processes that participate in an `MPI_Bcast` collective specify the same *root* argument. However, in the example both processes specify distinct values for this argument. The defect in the example is easy to understand and remove, but complex collectives like `MPI_Allgather` impose more complex restrictions that involve array arguments.

Process 0	Process 1
if $(x! = 5)$	`MPI_Barrier`()
`MPI_Barrier`()	
else	
`MPI_Bcast`(root:0)	

Figure 2.4: *For x as an input argument, this example highlights a control flow choice for process* 0 *that causes an input dependent collective mismatch defect.*

Process i
$\text{count} = 10 - \text{numProcs}$
request = `MPI_Isend` (to:$(i+1)\%\text{numProcs}$, count:*count*)
`MPI_Recv` (from:$(i-1)\%\text{numProcs}$)
`MPI_Wait` (&*request*)

Figure 2.5: *For* numProcs *as the number of application processes, this example highlights a process count dependent defect that manifests as a negative* count *value with more than* 10 *processes.*

The examples in Figures 2.4 and 2.5 highlight that the presence of failures may depend on input data or the number of application processes. The example in Figure 2.4 is correct as long as the input variable x is not 5, and exhibits the same failure as the example in Figure 2.2 otherwise. In practice, MPI applications may use different techniques for their data exchanges, depending on input size, chosen options, or the number of processes in use. As an example, an application may use a point-to-point implementation for a data exchange for up to 16 processes and an implementation with collectives for more processes. Such choices can result from in-depth performance investigations. The existence of such control flow choices complicates program verification, especially for runtime approaches.

The example in Figure 2.5 uses a variable number of processes (numProcs) and exhibits an MPI usage error (in the form of a negative *count* argument) when numProcs exceeds 10. This example demonstrates the impact of the process count for program verification.

Process 0	Process 1
`MPI_Recv`(from:1)	`MPI_Recv`(from:0)

Figure 2.6: *A basic deadlock situation that involves two interlocking receive operations of MPI.*

The example in Figure 2.6 illustrates a basic deadlock [87] scenario. Both processes issue a receive operation that only returns once a matching send operation becomes available. Since no such operation is available, both processes will wait indefinitely, i.e., they deadlock.

Process 0	Process 1	Process 2
`MPI_Send`(to:1)	`MPI_Recv`(from:ANY)	`MPI_Send`(to:1)
	`MPI_Recv`(from:2)	
`MPI_Barrier`()	`MPI_Barrier`()	`MPI_Barrier`()

Figure 2.7: *Depending of runtime choices of the MPI implementation, this example either runs correctly or it deadlocks.*

Whether some deadlocks manifest depends on non-deterministic choices such as in the example of Figure 2.7. All operations can complete if the send of process 0 matches the wildcard receive of Process 1, which can receive a send operation from any process. If the send of process 2 matches the wildcard receive of process 1 instead, then the example deadlocks since no receive becomes available for the send of process 0, no send becomes available for the second receive of process 1, and only process 2 is able to execute the `MPI_Barrier` operation. This deadlock is more challenging for usage error detection approaches, since it only occurs in some runs and since it involves point-to-point MPI operations as well as collective MPI operations.

Process 0	Process 1
r =`MPI_Isend`(to:1)	
`MPI_Wait`(r)	
`MPI_Finalize`()	`MPI_Finalize`()

Figure 2.8: *This example initiates a point-to-point send operation, but fails to provide a matching receive operation for the send. Performance considerations of most MPI implementations silently render the send operation as unmatched and uncompleted. Lack of user feedback complicates identification of this defect.*

Finally, Figure 2.8 illustrates a so-called lost send situation that can cause resource exhaustion failures, deadlock, or may not result in a failure. The `MPI_Isend` operation of process 0 initiates a nonblocking communication. The subsequent completion operation then attempts to complete the send transfer. It may return the control flow even if no matching receive operation is available, since the MPI implementation may use internal buffering for the send operation. Thus, when process 0 issues its `MPI_Finalize` operation to end its MPI usage, it can still have a pending send operation. The lack of an uncompleted MPI request highlights that such pending point-to-point transfers can easily remain unnoticed by an application developer.

2.4 Automatic Verification Methodologies

Existing approaches to automatically detect MPI usage errors use three methodologies: *Static program analysis*, which directly operates on the source code of an application; *Model-based formal verification*, which applies formal methods to a model of an MPI application; and *Runtime Verification*, which operates on information that was captured during an application run. This section introduces these method-

ologies and highlights their strengths and weaknesses. Since no methodology supersets the remaining ones, some tools and approaches combine multiple methodologies in order to combine their strengths.

2.4.1 Static Program Analysis

Static program analysis uses the source code of an application as input and then applies correctness analyses to individual statements or to more complex models that derive from this input. In the MPI verification scope, these analyses check against a specification that could be user provided or derived from the MPI standard. If a static analysis approach involves a model, it represents a combination with a model-based verification method (subsequent section). Additional source code annotations can provide supplemental input to refine the analysis. Existing approaches for MPI applications include a symbolic execution framework named TASS [143, 144], pCFG [20] as an extension for Control Flow Graphs (CFGs), barrier matching [90, 131, 174] methods, and basic MPI related correctness analyses such as in MPI-CHECK [108].

TASS uses symbolic execution to detect MPI related usage errors. This approach associates symbolic expressions to all input values and then represents the state of the program with these expressions. TASS enables an application input independent correctness analysis. This approach can detect defects that only manifests for some inputs, such that the tool user does not need to provide an input data set for the application to be verified. This facilitates the removal of defects that only manifest for some inputs. A further advantage of the symbolic execution is support for comparative symbolic executions that allow a comparison of a sequential and an MPI parallel implementation of the same application. TASS can reliably detect defects, such as for the example in Figure 2.4 (page 9), since it will consider two distinct control flow paths depending on the value of x. The complexity of the symbolic expressions and the total number of states that the approach considers during its enumeration of all potential control flow paths of a parallel program limit the applicability of TASS. As an example, if the approach fails to evaluate whether a complex symbolic expression can be satisfied, it may analyze a control flow path that an application could never take. Such an analysis could then reveal false-positives. The authors of TASS present results for a few small MPI applications [144].

The pCFG approach extends Control Flow Graphs (CFGs) to parallel message-passing applications. The approach uses an execution model that represents a subset of the MPI standard. Wildcard receive operations form one limitation of the approach. Correctness analyses on pCFGs can reveal matching failures, e.g., unmatched point-to-point operations or an MPI datatytype mismatch. This technique allows a process count independent analysis, which simplifies the detection of defects that only occur for some process counts, e.g., examples such as in Figure 2.5 (page 9) that exhibits a failure for more than 10 processes.

Barrier matching approaches analyze an application's source code to determine which collective operation invocations can match at runtime. If the analysis is successful, it can detect collective mismatch failures such as in the example of Figure 2.2 (page 8). This approach may report false positives due to assumptions that facilitate the source analysis, e.g., assumption of *structurally correct code* for which a study details counter examples [3].

Finally, MPI-CHECK uses Fortran 90 compiler capabilities to detect defects that can result from limited type checking capabilities of early Fortran versions.

2.4.2 Model-Based Formal Verification

Formal approaches use a model to verify correctness properties. The approaches MPI-Spin [141], ISP [156], DAMPI [162], and DPS [133] use this methodology to detect defects in MPI applications.

MPI-Spin extends the Promela language of the SPIN [81] model checker with relevant additional constructs for MPI. In particular, this includes constructs for nonblocking communications that extend the abstractions for channel-based communication in Promela. An MPI-Spin model then allows the SPIN model checker to investigate all states of the model in order to guarantee properties such as freedom from deadlock. This enables an execution independent detection of the interleaving dependent deadlock as in

the example of Figure 2.7 (page 10). The limited coverage of MPI constructs that MPI-Spin provides—no MPI communicator support, no collectives, only standard mode sends [140]—is a downside of this approach. Additionally, the model must be manually extracted from an application, which introduces additional opportunities for defects. Finally, the state space that SPIN needs to investigate may render the exhaustive state exploration too time consuming. Efficient models can reduce the state space, but require experience in the modeling process.

The approach in ISP also follows model checking, but automatically extracts the model during application runs, i.e., uses runtime verification (following section) in addition. ISP detects MPI constructs that can introduce non-determinism and enumerates all interleavings that these constructs can yield. The tool executes the application once for each interleaving and replaces the non-deterministic MPI constructs with rewritten deterministic constructs that enforce a specific interleaving. ISP uses the following assumption: The communication pattern of the application only depends on its input, its process count, and the return values of MPI operations, but no further non-deterministic values, e.g., random numbers. This assumption allows the tool to enumerate all interleavings with repeated executions, since similar interleavings will always exhibit the same trace of MPI operations. As a result, ISP can also reliably detect the deadlock in Figure 2.7 and it avoids the need to manually create a model. However, the search space to cover all interleavings of an application may easily render this approach too time consuming, even for very simple use cases of MPI [76].

DAMPI is a modified version of ISP that also extracts a model from application runs. However, it targets increased scalability. Where ISP uses a centralized scheduler to detect alternative interleavings and to determinize each execution, DAMPI uses a distributed approach to detect alternative interleavings. It uses a single execution of the application and applies Lamport clocks [103] to detect all alternative interleavings during that execution. Since Lamport clocks offer limited precision [44, 112], the approach may not be precise, i.e., can yield false positives or false negatives. Finally, once DAMPI identified individual interleavings in its single execution, it enumerates them and stores them for investigation. Subsequent executions of the application than use a decentralized scheduler that enforces the individual interleavings. A timeout-based deadlock-handling scheme then serves to detect deadlocks for each interleaving.

DPS uses state graphs to detect incorrect application behavior and considers their use for MPI. Like ISP, it uses reduction techniques to reduce its search space. However, it uses information on subsequent communication operations to apply additional reductions. In summary, MPI-Spin, ISP, DAMPI, and DPS can detect schedule dependent deadlocks such as in the example of Figure 2.7 (page 10). At the same time these approaches can lead to a state explosion that limits their applicability, e.g., see [56, 76]. Partial order methods can reduce the state space and can also be applied to the verification of MPI applications [155], but they can still fail to limit the state space sufficiently for practical verification purposes. As a result, the approaches MPI-Spin and ISP—which do not limit their precision as for DAMPI—currently report results for smaller sets of applications that make limited use of non-determinism, consist of few source code lines, or use few processes only.

2.4.3 Runtime Verification

Runtime verification approaches execute an instrumented MPI application, observe MPI operations at runtime, and then apply correctness analyses to this data. Figure 2.9 sketches this workflow. It uses circles labeled with numbers (0–3) to represent MPI processes, i.e., the example uses four application processes. Boxes with labels highlight that an instrumentation component observes MPI operations—and possibly additional information—on the application processes and that a correctness analysis applies to this information. The correctness analysis provides a correctness report to the tool user to summarize MPI usage errors, as well as potentially suspicious behavior. These runtime approaches are the main focus of this thesis and the following sections detail their advantages and limitations, as well as their architectures and scalability limits.

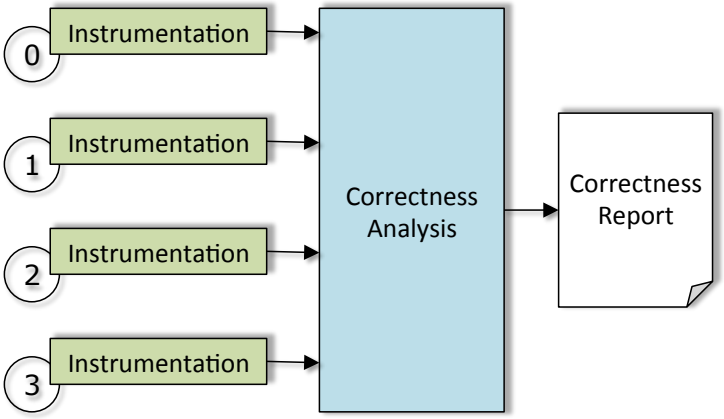


Figure 2.9: *An illustration of the overall runtime verification workflow for an MPI application with four processes, represented as labeled nodes. An instrumentation component observes the input for the actual correctness analysis, which reports failures to the user of the tool.*

2.5 Runtime Verification of MPI Applications

Runtime verification approaches apply a correctness analysis on data that they observe while an application executes. This analysis can follow a formal or an implicit model. The following also refers to correctness analyses as *correctness checks* or just *checks*. Often synonymously used terms for runtime verification include *correctness checking*, *runtime checking*, *dynamic verification*, and *dynamic testing*. This document uses the term *runtime verification* to classify this methodology and the term *runtime (MPI) correctness* approach/tool to refer to representatives of this methodology. This section first highlights the advantages and disadvantages of runtime verification approaches and then details individual representatives for MPI applications.

2.5.1 Tradeoffs

Non-deterministic control flow decisions can alter the behavior of an application at runtime. Also, application behavior can differ depending on the number of processes that an application run uses, or its input. Thus, runtime verification approaches check for the presence of failures in the interleaving that they observe during a specific application run. The absence of failures in such an interleaving does not indicate the absence of failures in other interleavings. While static program analysis techniques can apply their correctness analysis for any number of MPI processes or any input values, pure runtime verification approaches do not support such coverage. In addition, the runtime verification approaches in this section use the MPI standard as their specification, i.e., failures here represent violations to a restriction in the MPI standard. Model checking approaches, as an example, directly support user provided correctness requirements.

However, runtime verification approaches may offer higher applicability, e.g., they do not require application developers to provide a model of the application as in MPI-Spin [141], can support more MPI calls than ISP [155], can avoid assumptions that apply to the use of MPI calls as in DAMPI [162], and can also avoid exhaustive explorations of large state spaces. Also, the limited coverage allows for scalable approaches that support application developers at their target scale, which is a key motivation for this thesis.

Finally, runtime verification tools can apply static program analysis or model-based formal verification in addition, in order to increase coverage. Examples of such combinations include MPI-Check, ISP, and DAMPI.

2.5.2 Approaches

The following description of existing runtime MPI verification approaches uses a structuring with three approach groups: *Focus on communication buffer related restrictions; Focus on restrictions that are specific to collectives; And wide coverage of restrictions*. These groups highlight approaches with largely similar functionalities.

Additionally, some related runtime approaches use the information that they observe to reveal potential failures rather than to verify against specific requirements. This includes MPIRace-Check [122], AutomaDeD [102], and DIDUCE [62, 105]. The former focuses on the detection of non-determinism, irrespective of whether it violates a specification of the MPI standard or not. Approaches such as AutomaDeD apply clustering and similarity algorithms on state information of an MPI application. This data then identifies processes that behave abnormally, i.e., different from most processes. The approach in DIDUCE uses runtime information to automatically create hypotheses based on observed behavior. The tool then reports violations to these hypotheses in subsequent application runs to reveal potential failures.

2.5.2.1 Communication Buffer Usage

The MPI standard defines that applications must not write to memory regions that previous and uncompleted send operations use as their communication buffers. In addition, an application must neither write nor read memory regions that previous and uncompleted receive operations use as their communication buffers. This both includes a direct memory access by the application and passing memory regions to libraries that could access the memory in a reading/writing manner. Particularly, the latter includes further MPI operations that could access parts of an active communication buffer. Violations to this restriction will usually not manifest in a crash or hang and may remain unnoticed as a result. Additionally, the presence of many pending communications or complex memory movements, e.g., all-to-all exchanges in collective operations, makes it hard to track data regions that are in use as communication buffers. Approaches such as SyncChecker [29] and OpenMPI-Ext [42, 43, 94] (an extension of OpenMPI) target this specific requirement. These approaches observe MPI communication operations and information on read or write accesses to communication buffers. Technologies such as Pin [9] or Valgrind [118] can provide the latter information. Both options will heavily influence the overhead of the tool, as well as its availability for different compute platforms.

FlowChecker [28] is a related approach that detects failures within MPI implementations. It verifies that communication operations actually implement the memory movements that the MPI standard describes. As a result, FlowChecker aids in the development and verification of MPI implementations.

2.5.2.2 Collective Usage

Both MPI/SX [153] and MPICH-Coll [40] (a library of the MPICH MPI implementation) provide correctness checks for MPI collective operations. These approaches inject additional MPI collective operations prior to each collective that they observe. The additional operations then provide the means of communication to detect violations to collective related MPI specifications. As an example, to detect the inconsistent root argument in the example of Figure 2.3 (page 9), injecting an `MPI_Reduce` before the incorrect `MPI_Bcast` operation enables the detection of the inconsistency. The number of collectives to be inserted depends on the type of the collective that is being verified. MPI/SX does not consider all specifications for collective operations, since it provides no type matching checks. However, MPICH-Coll overcomes this limitation and uses a hash-based type signature comparison [58] to detect type matching failures. This hash-based technique is simple to implement, but can cause false negatives at the same time. In addition, both MPI/SX and MPICH-Coll do not detect or otherwise handle deadlock.

2.5.2.3 Wide Coverage

The approaches ISP [156], DAMPI [162], Intel Message Checker (IMC) [132], Intel Trace Analyzer and Collector (TAC) [120], Marmot [99], MPIDD [64], and Umpire [159] all detect wide ranges of violations to the MPI standard. This includes the detection of incorrect arguments, MPI resource usage failures, type matching failures, and deadlocks. The example failure in Figure 2.5 (page 9; for more than 10 processes) illustrates an incorrect argument in the form of a negative integer—where a positive integer within a certain bound is expected. MPI offers various resources to manage process groups, datatypes for communication operations, and pending communications. Many types of failures can result from the creation, use, and de-allocation of these resources. However, checks for these failures involve information from a single process only, and thus, have less impact on the scalability of the individual approaches. The functionality classes of type matching and deadlock handling involve tool internal communication in order to exchange information from multiple application processes. Thus, directly impacting tool scalability. The example of Figure 2.1 (page 8) illustrates a type matching defect while the example of Figure 2.6 (page 10) illustrates a deadlock.

The approaches ISP and DAMPI focus on deadlocks, resource usage errors, and type matching; where they use a model-based formal verification approach in addition (previous section). This allows the two approaches to enumerate and test alternative interleavings of an MPI application. DAMPI uses a timeout-based deadlock handling that offers no dependency analysis and that may report false positives. ISP in comparison, implements a transition system [155] that indicates deadlock as a state to which no transition rule is applicable. This approach avoids false positives, but only provides limited visualization and analysis capabilities to investigate a deadlock situation.

Both IMC and TAC are proprietary tools that operate with a single MPI implementation only. IMC uses a trace-based approach that stores information on MPI operations in a trace file during the application run. An offline analysis reads and processes the trace file after the application run to detect MPI usage errors. The tool TAC supersedes IMC with a pure runtime approach, where tool developers considered the trace approach as a limitation [120]. TAC provides checks for violations to a wide range of MPI specifications. Particularly, it detects resource usage errors, combines MPI piggybacking [134] with a hash-based signature comparison approach [58] to compare MPI datatypes, and uses additional collective operations to verify MPI collectives as in MPI/SX and MPICH-Coll. Timeout-based deadlock detection handles deadlocks in TAC and comes with the same disadvantages as the timeout approach in DAMPI.

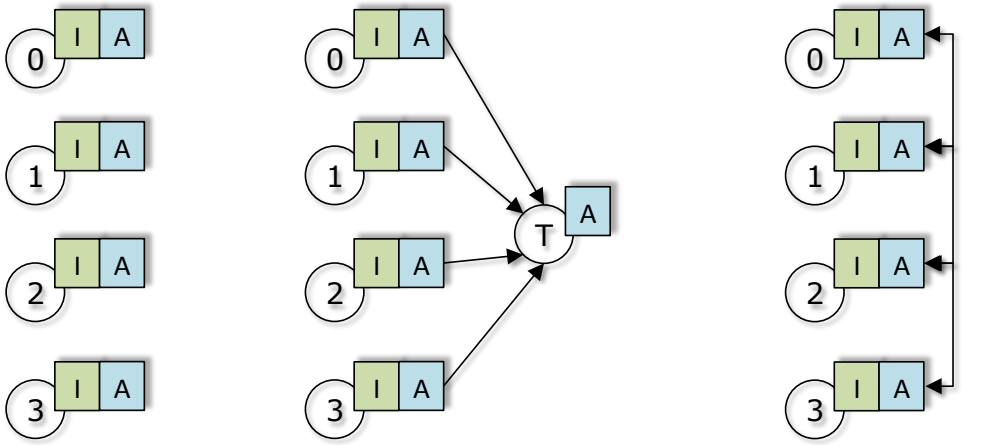
The tool Marmot targets checks for invalid arguments, MPI resource usage, and provides timeout-based deadlock detection. However, it does neither support type matching nor checks for collective operations.

MPIDD targets deadlock detection with a dependency analysis that uses a cycle as a necessary and sufficient deadlock criterion. Complex wait-for semantics of some MPI operations can imply large recursive search spaces with such a cycle search [69].

Finally, Umpire provides few checks for invalid arguments, but detects MPI resource usage failures, type matching failures, as well as deadlocks. For the latter two, Umpire transfers information on observed MPI operations from all processes to a single extra tool thread. This thread uses a type signature comparison [125] for the detection of type matching failures. This algorithm neither provides false positives nor false negatives. The deadlock detection capability in Umpire [69] tracks dependencies of MPI operations in a wait-for graph and uses a graph criterion to reveal and visualize deadlocks. Particularly, this analysis allows Umpire to discern processes that cause a deadlock from processes that hang as a result of the deadlock situation.

2.6 Tool Architectures

The tool architecture of a runtime correctness tool influences both its scalability and its applicability. As examples, a performance bottleneck in a tool can limit its scalability and assumptions on communication service availability can limit its portability. This section categorizes the architectures of the runtime



(a) Purely local. (b) Centralized. (c) Inline communication.

Figure 2.10: *Illustration of architecture types for existing MPI runtime verification tools. The illustration highlights options to distribute instrumentation ("I") and correctness analysis ("A") components onto application processes (0–3) and additional control flows ("T").*

correctness approaches from the last section into three types: *purely local*, *centralized*, and *inline communication*. Figure 2.10 illustrates these architecture types. The illustration represents application processes as labeled circles, along with instrumentation and analysis components that use the colors of Figure 2.9. However, Figure 2.10 shortens the labels of the instrumentation component to "I" and the label of the components for correctness analysis to "A". The association of the components to circles indicates which processes execute them. Additionally, the centralized architecture in Figure 2.10(b) uses an additional tool *place*, which is a tool specific process or thread that executes a part of the correctness analysis. The figure highlights this tool place with a circle that uses the label "T". The arrows in the figure indicate tool communication to forward information from instrumentation components to correctness analyses that run on other application processes or on a tool place.

2.6.1 Purely Local

SyncChecker and OpenMPI-Ext both detect invalid read/write accesses to active communication buffers. The approaches execute this correctness analysis directly on the application processes, since the available information on each process suffices for this analysis. These approaches do not require any tool internal communication and can use the purely local architecture type from Figure 2.10(a). A purely local tool imposes no scalability limitation if the cost of its instrumentation component and its correctness analysis matches the cost of the operations that they observe and analyze. However, such a tool must provide a correctness report to the user. If each application process creates one file for its correctness report then I/O system limitations [84] can still restrict scalability.

2.6.2 Centralized

Some correctness analyses require tool internal communication, e.g., the example MPI usage error in Figure 2.1 (page 8) requires that a correctness tool combines information on the datatypes of a sending and of a receiving process. The centralized architecture type (Figure 2.10(b)) uses an additional tool owned process or thread, called a tool *place* to execute correctness analyses that require information from more than one process. Figure 2.10(b) represents the tool place with a node labeled "T". For the previous datatype mismatch example (Figure 2.1 on page 8), a tool with a centralized architecture would send information on the type signatures of the send and receive operations of processes 0 and 1 to the

tool place. Once the place receives this information from both processes it can compare the signatures and detect the failure.

MPIDD, Marmot, Umpire, ISP, and FlowChecker all use the centralized architecture type. However, their realizations of both the tool place and of the tool internal communication differ. Umpire, for example, uses an extra thread on one of the application processes as a tool place and either employs MPI or shared memory for its tool internal communication. ISP, as a second example, uses a sequential program that runs parallel to the application as its tool place and uses TCP sockets for tool internal communication.

A single tool place allows tools to execute correctness analyses that require communication. If the rate at which application processes issue MPI operations remains constant or even increases with scale, then the workload on the single tool place also increases with scale. Thus, the centralized place is a scalability bottleneck and limits the scalability of the approaches that use this architecture type. The single tool place usually also provides a single correctness report to the tool user.

2.6.3 Inline Communication

The tools MPIRace-Check, MPI/SX, MPICH-Coll, MPI-CHECK (for deadlock detection [110]), DAMPI, and TAC use the inline communication architecture type. Tool injected MPI operations—except for TAC that uses an MPI implementation internal API—distribute input for correctness analyses in these approaches. This enables a direct utilization of MPI as a means of communication and can simplify tool implementation and deployment. In addition, it allows these approaches to directly use the application processes to execute correctness analyses that require communication. TAC, MPI/SX, and MPICH-Coll inject additional MPI collective operations when they observe a collective operation on an application process. The approaches in MPIRace-Check, DAMPI, and TAC use piggybacking [134] methods to attach information for their correctness analyses to point-to-point operations.

Inline communication allows these tools to scale with the number of application processes. Further, one of the processes can create a single correctness report. The approaches MPI-CHECK, DAMPI, and TAC handle MPI deadlock, where all three tools use a timeout approach. A graph-based approach or the use of a transition system can offer increased detail into failure situations and avoid false positives. However, the use of inline communication for these techniques is challenging, since the tool must be able to detect and report a deadlock before it manifests, in order to utilize MPI as the tool internal communication medium. Thus, the inline communication would need to distribute the information that the tool requires to detect a deadlock, while at the same time this communication must not cause deadlock in the first place.

2.7 Comparison

The following compares the runtime verification approaches from the last section. The authors of MPI/SX [154] made a first comparison of a subset of these approaches to highlight the strength of their approach, while they could not conclude that a single approach is superior to others. Authors of ISP and TAC extended this comparison [120, 138] to draw two conclusions: First, TAC provides a wide variety of correctness checks, while it also offers the ability to scale; Second, added coverage from an analysis of alternative interleavings, such as in ISP, aids in the development of correct programs.

Most of the runtime tools that this section introduced have at least one unique feature. Thus, a tool comparison will not identify a single tool that supersedes all other ones. As an example, TAC may provide a wide range of checks and include the technology of MPI/SX [154] to efficiently handle collective operations. However, Umpire provides fewer checks, but provides deadlock detection with a graph-based scheme rather than just a timeout algorithm. Also, Umpire supports precise datatype comparisons where TAC uses a hash comparison that can lead to false negatives. ISP on the other hand improves on the deadlock detection capabilities of Umpire in the sense that it automatically explores multiple (all) interleavings of an MPI application. As an example, Umpire only detects the deadlock in the example of

	Known Application Scale		Architecture	Collectives	P2P	Type Matching	Deadlock
DAMPI	1,024	[162]	Inline				✓ (Interleavings + Timeout)
TAC	256	[120]	Inline	✓	✓	✓ (Hash)	✓ (Timeout)
MPICH-Coll	32	[41]	Inline	✓	×	✓ (Hash)	×
Umpire	512[1]	[69]	Central	✓	✓	✓ (Regexp)	✓ (Graph-based)
Marmot	16/128[2]	[96, 71]	Central	×	×	×	✓ (Timeout)
ISP	8	[137]	Central		✓	✓ (Undoc)	✓ (Transition System)
MPIDD			Central				✓ (Implicit Graph)

Table 2.1: *A comparison that summarizes whether and how existing MPI runtime verification approaches support correctness analyses that challenge scalability.*

Figure 2.7 (page 10) if it manifests in the run with the tool, while ISP would automatically analyze both possible interleavings in order to reliably detect the deadlock.

Thus, this section identifies key properties that distinguish the individual tools for their use at scale, which is the focus of this thesis. Also, these properties motivate novel methods that subsequent chapters introduce.

2.7.1 Properties

Correctness analyses that require tool internal communication can limit the scalability of a runtime correctness approach. The following properties specifically include the correctness functionality that requires communication, since this thesis targets increased scalability for correctness tools:

Type matching: Checks whether two type signatures match according to the rules in the MPI standard,

P2P: Matches point-to-point operations (preprocessing for type matching, deadlock analysis, and the detection of lost send operations as in Figure 2.8 on page 10),

Collectives: Matches collective operations (preprocessing for type matching and to detect violations to collective related MPI specifications such as the root mismatch of Figure 2.3 on page 9), and

Deadlock: The handling of deadlocks.

Support for these types of analyses serve as the first four properties to compare runtime correctness approaches. These properties explicitly exclude features with a limited impact on scalability, such as: The ability to detect invalid memory accesses to active communication buffers (e.g., SyncChecker), invalid argument checks (e.g. Marmot/TAC), or the detection of MPI resource usage errors (e.g., Umpire/ISP/Marmot). The tool architecture type serves as the fifth property since it impacts whether a tool can scale in general. Finally, the last property is the demonstration of scalability in application studies, since scalability limitations may also exist for tool architectures that pose no scalability bottlenecks.

2.7.2 Categorization

Table 2.1 compares runtime correctness approaches for the selected properties. The "Known Application Scale" column lists the highest known scale for which the tool could analyze an application, along with a reference to the respective study. The "Architecture" column specifies the architecture type of

[1] As a trace-based offline analysis
[2] Only process local correctness analyses

each approach as “Inline” for the inline communication architecture and as “Central” for the centralized architecture. The remaining columns use “✓” to indicate that an approach supports a functionality class and “✗” to indicate that it does not. Additionally, braces can detail the techniques that an approach uses to provide correctness functionality.

The table excludes SyncChecker and OpenMPI-Ext, since these approaches only consider process local correctness analyses that have limited scalability impact. Further, it only lists MPICH-Coll since it supersedes the functionality of NEC/SX. Finally, the comparison excludes IMC since TAC intends to supersede this approach. Empty cells indicate that no information on a property was available for a certain approach, e.g., the table lists no known application scale for MPIDD, since no publication or report presents application results for this approach (neither was its source code openly available when this document was written).

The comparison highlights that all approaches fall into one of the following two categories:

Scalable, not precise: The approach offers scalability, but can report false positives or false negatives, or

Precise, not scalable: The approach handles deadlocks and/or type matching without false positives or false negatives, but limits scalability.

The only exception to this categorization is Marmot, which provides a wide range of process local correctness checks and a variety of tool integrations for increased usability. Marmot’s very limited support for correctness analyses that involve multiple processes forms a part of the initial motivation for this work.

The tools DAMPI, TAC, and MPICH-Coll fall into the category *scalable, not precise*. DAMPI uses the scalable inline-communication architecture type and demonstrates scalability with up to $1{,}024$ application processes. However, from its available information, it only handles MPI deadlock. It can detect alternative interleavings, but when it analyses a particular interleaving it uses a timeout-based handling that can cause false positives and that provides no dependency details on the deadlock situation. TAC also uses a timeout-based deadlock handling and provides hash-based type matching that can cause false negatives. MPICH-Coll does not handle deadlocks and only considers collectives, where it also uses a hash-based approach to implement MPI type matching checks.

On the other hand, Umpire, ISP, and MPIDD use tool architectures that limit scalability, but avoid false positives or false negatives for the correctness analysis that they provide. ISP provides type matching with an undocumented implementation and detects deadlocks. The lack of an applicable transition in a state of a transition system indicates deadlock [155]. Both MPIDD and Umpire use a graph-based deadlock detection. Umpire uses a necessary and sufficient deadlock criterion [69] that suffices for all MPI wait-for dependencies, while MPIDD uses a graph search that tries to handle wildcard receives with a recursive cycle search. In addition, Umpire also provides a type signature comparison that avoids false negatives and false positives for type matching checks.

2.8 Summary and Goals

The comparison of existing runtime correctness tools highlights that both scalable and precise approaches exist. However, no single tool in the comparison achieves scalability without sacrificing precisions, i.e., without the use of timeout-based deadlock handling techniques or the use of hash-based MPI type comparisons. As a consequence, this thesis investigates concepts that could fill the currently empty category:

Scalable and precise: The approach offers scalability with correctness analyses that avoid the use of techniques that can introduce false positives or false negatives.

As to limit the scope of the thesis, the four mentioned correctness analyses that require tool internal communication (type matching, point-to-point matching, collective matching, and deadlock detection) serve as demonstrators to design such an approach.

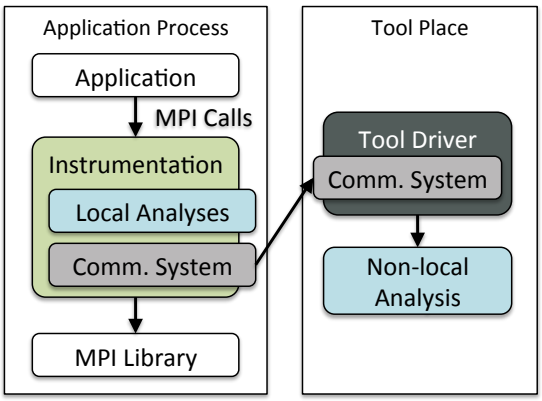


Figure 2.11: *Illustration of common tool components that enable offloading-based runtime verification approaches.*

Table 2.1 also highlights that all approaches that provide precise deadlock detection use a centralized tool architecture. The use of the inline communication architecture type is challenging for such tools [40], since the tool internal communication must both distribute data for deadlock detection prior to deadlock manifestation and must not cause deadlock itself. Tools that use a centralized tool place *offload* information on MPI operations to the additional tool place for correctness analysis. This offloading technology simplifies deadlock detection, since the tool place has its own control flow, which will not be impaired by a manifest deadlock on the application. The following refers to approaches that offload information for correctness analysis to additional places with their own control flow as *offloading*-based tools or approaches.

Figure 2.11 illustrates that offloading-based tools have multiple common components. This includes an instrumentation component that observes MPI operations and passes their information to correctness analyses that can directly execute on the application processes, as well as to a communication system that offloads information. A main loop called *tool driver* in Figure 2.11 must handle the control flow on tool places. It must receive information on MPI operations and pass it to correctness analyses that run on the tool place. The figure refers to correctness analyses that do not require tool communication as *local analyses* and to correctness analyses that require tool communication as *non-local analyses*. Thus, offloading-based tools require components for instrumentation, a communication system, and a tool driver. Tool developers can invest considerable amounts of time in creating these components themselves. Such custom-made tool components can then limit component reuse and the composability of a tool. Parallel tool infrastructures—as the following chapter introduces—can provide well tested and widely applicable tool components to both reduce the development time for tools and to reuse investments into portability and scalability.

3 State-of-the-Art in Parallel Tool Infrastructures

> *The lack of a standard for HPC tool infrastructure has resulted in myriad implementations, but few have addressed the scalability and portability demands that these new systems will place on tool developers.* [24]

This chapter analyzes the applicability of parallel tool infrastructures for MPI runtime verification. The usage of such infrastructures provides advantages during tool development and maintenance (Section 3.1). Section 3.2 then summarizes functionality classes that parallel tool infrastructures should provide for MPI runtime verification. Afterwards, Section 3.3 presents how existing tools or tool infrastructures provide these functionality classes. Tree-Based Overlay Networks (TBONs) can provide abstractions for scalable tools, where Section 3.4 presents these abstractions and discusses their applicability for MPI runtime verification. Finally, Section 3.5 summarizes the applicability of existing parallel tool infrastructures for MPI runtime verification and motivates their advancement.

3.1 Motivation and Terms

Most offloading-based runtime tools use similar types of components [32, 52, 115, 129, 136]. Several tools for runtime MPI verification, as well as tools for other use cases, provide their own implementations for most or all of these components, i.e., they use *custom components*. Such a development approach complicates component reuse, due to the use of specialized interfaces. Additionally, tool composability towards efficient component reuse and tool integration is challenging for custom components, e.g., coupling two MPI tools [71]. Section 3.2 details this problem for the runtime MPI correctness tools Marmot and Umpire. The developers of custom components must address challenges such as portability across compute systems, increasing compute system scale, and support for novel programming paradigms themselves and can't directly reuse the solutions that other tools employ. This situation increases the development time and maintenance costs of runtime tools, as well as the confidence that application developers have into these tools [157]. At the same time, parallel tool infrastructures demonstrate that they can provide several types of common components to tool developers [24, 93, 49, 129]. Abstractions in these infrastructures provide concepts that simplify reuse of tool functionality across different tools. If these abstractions specifically consider parallel computing as the primary use case, they can also include scalability features that highlight that parallel tool infrastructures can efficiently support scalable tools [104].

Since tool infrastructures can support different types of tools, e.g., performance analysis or monitoring tools, the following will refer to any tool related computation as an *analysis* or *tool analysis*. In the case of a correctness tool, a tool analysis could be a correctness check; whereas for a performance optimization tool an analysis could be the task of adding a new performance relevant event into a trace buffer. The term *event* refers to information that an analysis requires as an input. For correctness checks, an event could be information about the occurrence of an MPI operation. Application processes usually create events that may be communicated within the tool to pass them to the location (place) where a tool analysis requires them. However, tools may also inject their own events to implement their functionality. In addition, tool infrastructures can distinguish from tool frameworks in the depth of their abstraction. That is, the overall control may be with the tool (tool calls infrastructure services) or the framework (framework calls tool analyses). The following will not distinguish between these two concepts and refers to both with the term *infrastructure*. The following sections will highlight situations in which a framework approach provides an advantage or disadvantage.

	Marmot	Umpire
Instrumentation	Script that generates wrappers for MPI calls	Generated from an input that uses a specification language to describe the handling of each MPI call
Topology	Centralized (extra process)	Centralized (extra thread)
Means of Communication	MPI with a communicator virtualization	Purely shared memory or combination with MPI
Handling an application crash	Synchronizing communication guarantees that events are analyzed before calls are issued to the MPI library	Signal handlers, at-exit handlers, and MPI error handlers with threads
Flow and Abstractions	Hardcoded behavior on application processes and the single tool place. Generated instrumentation uses Marmot specific interfaces to start local analyses and to communicate MPI call information	Data flow and applicable correctness analyses result from specification language

Table 3.1: *This component summary of the MPI runtime verifications tools Marmot and Umpire highlights how a custom-made development approach complicates tool integration and component reuse.*

3.2 Requirements for MPI Runtime Verification

Figure 2.11 highlights key components for offloading-based runtime tools with an MPI use case. Excluding the tool specific analyses—e.g., correctness checks for runtime verification tools—such a tool requires the following services:

Instrumentation: Provides capabilities to observe relevant events, such as an MPI call being triggered;

Topology: Defines the architecture of the tool including tool place(s) and their communication connections;

Means of Communication: Pairs of application processes and or tool places that the topology connects must use a means of communication to exchange information;

Crash-Handling: Runtime verification considers potentially incorrect applications that may fail with a crash, a crash-handling technique must ensure that a tool detects a usage error even if it causes an application crash;

Flow and abstractions: Abstractions define the data flow between components—e.g., between instrumentation and local correctness analyses—and how analyses are loaded onto the topology of tool places.

A runtime correctness tool is required to employ some technique or component for each of the above services. The following considers these services as requirements for a correctness tool. Individual implementations for these services then influence non-functional [31] requirements such as scalability, maintainability, ease-of-use, composability, reusability, and portability. The following sections consider scalability as a key requirement that highly depends upon the tool topology. Further, reusability, pluggability, and composability largely depend on choices for the flow and abstraction service.

The following sections introduce existing parallel tool infrastructures and how they provide the above services. But first, the study in Table 3.1 summarizes the techniques that the existing runtime correctness tools Marmot and Umpire use to implement the above services with a custom component approach. This study serves to highlight the disadvantages of such a development workflow and to introduce first techniques to provide the required services for an MPI runtime correctness tool.

Marmot uses C++ classes to represent MPI calls and implements the actual tool analyses in methods of these classes. Umpire uses a tool specification language that defines which correctness analyses apply to which MPI call. The specification language allows Umpire to automatically generate source code that implements its data flow, whereas Marmot uses a hard coded flow in its hierarchy of C++ classes. At the same time, Umpire's specification language is highly tool specific, where Umpire's source code generator uses hard-coded handlers for its specifications.

Both Marmot and Umpire use a centralized topology with one tool place. Marmot uses an extra MPI process for this purpose, while Umpire uses an extra thread instead. An important notion is that the analyses of both tools make assumptions on their locality. That is, analyses executing on the application processes use interfaces that are only available on the processes, whereas analyses on the single tool place use interfaces that are only available on the tool place. As a result, in both tools, analyses assume the presence of tool specific interfaces and can't easily be decoupled. As an example, migrating an analysis that Marmot usually executes on the application processes onto its tool place is not directly possible. Thus, locality of analyses can't be changed in both tools, due to static assumptions; let alone is it possible to transfer analyses of the one tool onto the other—as would be very synergetic in the case of the two tools that implement largely disjoint sets of correctness checks. The example of these two tools highlights the limited software reuse that results from a tool development with custom components.

As an example for component reuse, assume that either of the tools develops a component that provides some service in an advantageous way, e.g., Umpire's crash handling technique works well in practice and comes with a limited performance impact (as subsequent sections illustrate). Integrating such a component into the other tool—from Marmot to Umpire or vice versa—is challenging since the custom made components of the tools use their own interfaces. Any integration requires component adaption or a bridging of interfaces. Such a situation limits the reuse of components across wide ranges of tools.

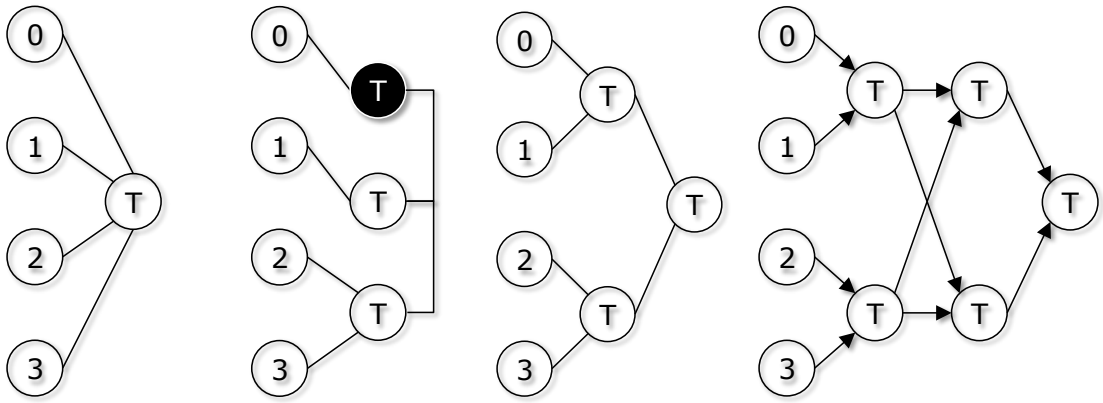
Tool infrastructures can efficiently revert this situation. If both tools used the same infrastructure, then analyses of both tools would rely on similar interfaces to connect to infrastructure services. This situation would drastically simplify functionality reuse between the tools. Similarly, if one tool improves a component of the infrastructure, then the other tool could benefit from this improvement with minimal adaption.

3.3 Existing Services and Infrastructures

This section summarizes existing technologies for each of the services that an MPI runtime verification tool requires. This specifically includes approaches from infrastructures, as well as from existing tools with custom components. However, the abstraction and flow service is highly tool specific for tools with custom components, as the previous section introduced for Umpire and Marmot. These tools usually employ a generator script (as in Umpire) or a hardcoded data flow (as in Marmot) that is specific to the individual tool. Thus, for the abstraction and flow service, this section focuses on parallel tool infrastructures and how they provide generic approaches to let tool developers integrate their analyses with individual infrastructure components or frameworks.

3.3.1 Instrumentation

MPI runtime verification tools must observe MPI calls that an application issues at runtime. The MPI standard provides a profiling interface [113] that allows so-called *wrappers* to intercept invocations of the interface. Given the size of the MPI standard, manual wrapper generation is laborious. As a result, most runtime MPI tools and infrastructures employ generators to create these wrappers. Examples include the generators in mpiP [158], in Marmot, and in Umpire. Both Marmot and Umpire generate their wrappers such that they match the tool specific interfaces and provide very limited flexibility for other use cases in their generators. The generator in mpiP uses a specification language such that tool developers can easily influence its output. The use of such wrapper generators is the most common technique to implement the instrumentation service in a pure MPI tool.



(a) Centralized topologies limit scalability.

(b) Distributed topologies enable scalability, but do not provide a notion of hierarchies.

(c) Tree Based Overlay Network (TBON) topologies enable both scalability and event aggregation/-filtering concepts.

(d) Directed Acyclic Graph (DAG) topologies enable scalability, but can impact the event order. Using an *away-from-application* arc direction.

Figure 3.1: *Common types of tool topologies provide immediate feedback on tool scalability.*

Additional information sources to which an instrumentation is applicable include: Wrappers for generic interfaces as in TAU [139] or VampirTrace [35]; Binary instrumentation, e.g., Dyninst [22]; And memory access instrumentation, e.g., Pin [9] or Valgrind [118]. The use case of a tool determines whether it requires some of these additional sources. The correctness functionality that this thesis considers does not require any of these sources and can purely rely on MPI instrumentation.

A tool infrastructure should at least provide automatic wrapper generation for MPI, where additional instrumentation sources can help for advanced tool functionality. The infrastructures P^{n}MPI [135] and OCM [170] both include an instrumentation service. The former uses the mpiP wrapper generator to provide instrumentation capabilities for MPI, while the latter uses a generic instrumentation system approach that supports predecessor paradigms of MPI, e.g., PVM [149].

3.3.2 Topologies

Figure 3.1 provides a classification of offloading-based runtime tool topologies with four categories. The illustration uses nodes to represent application processes (for p processes, labels $i \in \{0, 1, \ldots, p-1\}$) and tool places (label T) as in Figure 2.10. However, edges represent communication connections and do not highlight a common communication direction, except for Figure 3.1(d) that requires directed edges for its topology. The *centralized topology* in Figure 3.1(a) realizes centralized architectures of tools such as ISP, Marmot, and Umpire. These tools use the single tool place to execute analyses that require information from more than one application process. While often used for runtime MPI correctness tools, this topology limits scalability since the single tool place is a bottleneck.

Figure 3.1(b) illustrates the *distributed topology* that provides multiple tool places. Each tool place receives information from a sub-set of the application processes. Pairs of places can then directly communicate with each other in order to exchange this information. Additionally, one of the places may serve as a master place to define a master-slave hierarchy between the places (Inverted node in Figure 3.1(b)). This topology type is related to runtime verification approaches that use inline communication, such as NEC/SX or MPICH-Coll. In these approaches, application processes also serve as tool places, i.e., they share their control flow. Further tools with a distributed topology are VampirServer [21] and Scalasca [53]. The former operates post-mortem on a trace file and the latter operates either post-mortem or directly at the end of an application run. This topology allows tool developers to distribute analyses over multiple tool places in order to derive scalable analyses. The notion of a master place already highlights that some tool analyses may require a hierarchy for their distribution. If a tool, or a

tool infrastructure, uses a master-slave hierarchy, the master place can limit scalability.

Tree-Based Overlay Networks (TBONs) define a hierarchical topology—Figure 3.1(c)—that can support tool developers when distributing analyses. This topology type resembles the distributed topology, but it organizes its places in hierarchy layers instead. Use cases for such a hierarchy include merging and evaluating performance data, e.g., in Periscope [55] or TAUoverMRNet [117]; distributing commands and aggregating state information in a debugger e.g., Ladebug [12]; aggregating state information for stack trace analyses, e.g., STAT [7]; and launching parallel applications or tools, e.g., LaunchMON [1]. In particular, tool infrastructures exist that provide a TBON layout, e.g., SCI [93], STCI [24], and MRNet [129]. Where the latter provides TBON services for multiple existing runtime tools, including TAUoverMRNet, STAT, and LaunchMON. The next section then details the TBON concept and its applicability for runtime MPI verification.

Debugger approaches for parallel applications, such as Ladebug [12], SPDL [91], and Allinea DDT [5], also utilize TBON concepts for scalability reasons. Their use of aggregation and broadcasting services in such a topology closely relate to the MPI runtime verification use case. The report on Ladebug details how a parallel debugger can use TBON concepts efficiently. Allinea DDT uses a proprietary TBON implementation and does not offer an API to access this service. This chapter focuses on approaches that offer access to their TBON services and excludes parallel debuggers that do not offer such access, i.e., Ladebug and Allinea DDT. SPDL on the other hand, is able to utilize services of TBON-based tool infrastructures for its communication and is able to employ both MRNet and SCI. This example highlights that parallel debuggers can also utilize parallel tool infrastructures, rather than relying on proprietary interfaces.

DeWiz [100] uses a module communication graph abstraction to create tools. This abstraction yields a communication topology that forms a Directed Acyclic Graph (DAG), as Figure 3.1(d) illustrates. In DeWiz, each tool "module" implements a specific analysis, where the output of one module may directly be used as the input for another module. This concept realizes a dataflow programming concept [164]. The communication graphs of DeWiz must not include cycles, i.e., must be a DAG. A DAG topology provides a hierarchy along with a potential for additional parallelism in higher-level layers. At the same time, information can arrive on a place via different paths. This property influences analyses that require events to arrive in the order they were generated.

3.3.3 Means of Communication

Offloading-based runtime tools require a means of communication between pairs of application processes and/or tool places that the topology connects. For the task of transferring information, a single means of communication suffices, e.g., Marmot uses MPI communication, ISP uses TCP socket communication, and the infrastructure MRNet also uses TCP socket communication. However, if an infrastructures provides flexibility to choose a means of communication, then it increases tool flexibility and portability, e.g., Umpire uses a combination of shared memory and MPI-based communication. Support for multiple means of communication simplifies strategies for handling an application crash (subsequent section) and it provides alternatives for systems on which some means of communication are unavailable. The debugging library SPDL [91] demonstrates a high degree of flexibility with an exchangeable communication system that can even employ different tool infrastructures.

For the use case of MPI runtime verification, an infrastructure should provide a means of communication with bandwidth and latency that is comparable to the means of communication between application processes, since otherwise the tool communication system may become a bottleneck. This requirement results from the need to capture information for each communication operation of the application. The MPI profiling tool MALP [16] communicates profiling records for all MPI operations during the runtime of an application. For this communication it uses an MPI-based means of communication. The authors of this approach report that in practical cases they require about 10% of the bandwidth that their MPI-based communication provides. As a result, MPI runtime verification uses cases—that also analyze all MPI communication operations—will require a comparable fraction of the available MPI bandwidth.

3.3.4 Application Crash-Handling

Runtime MPI verification tools must be able to detect an MPI usage error even if it causes an application crash. The inline-communication approaches TAC, MPICH-Coll, and NEC/SX finish all correctness analyses before they issue an MPI call to the MPI library. With that they can detect an MPI usage error before it can cause an application crash. Marmot and ISP also uses this *analyze first, issue afterwards* scheme; but both need to communicate information on MPI calls to their central tool places in addition. These tools use synchronizing communication for this task and finish the communication to the central tool place, as well as any correctness analyses on that place, before a process issues a call to the MPI library. While simple, this scheme introduces a high degree of synchronization with the central tool place and causes high runtime overheads.

Umpire uses a purely asynchronous crash-handling scheme that enables the use of asynchronous communication. It uses an extra *outfielder*-thread on each process, i.e. one additional tool place per process. Since this extra tool place only serves for crash handling purposes, rather than for distributing tool analyses, the previous section classified Umpire into the centralized tool topology (Figure 3.1(a)). When a process issues an MPI call, it first transfers information on the call to the outfielder (via shared memory) and issues the call to the MPI library afterwards. If the call causes an application crash, a signal handler, an MPI error handler, or an at-exit handler catches this situation and notifies the outfielder. The outfielder then communicates all outstanding information to Umpire's central tool place (via MPI communication), which is a further extra thread on process 0. This scheme ensures that even if an application crash occurs, no information on MPI operations is lost and that the central tool place can analyze this information. However, this scheme uses the assumption that if an application process has a crash, then another thread of this process can still use MPI communication. The MPI standard does not require MPI implementations to guarantee such a property. Additionally, since Umpire uses MPI communication on multiple threads in parallel, it utilizes the highest level of thread support (`MPI_THREAD_MULTIPLE`) that an MPI implementation can offer. This support can incur a performance penalty [150], while MPI implementations may choose to not support this level of thread support in the first place.

The tool infrastructure MRNet provides failure recovery for tool places of a TBON layout [8], but does not support situations where a crash occurs on an application process. The authors of MRNet mention that they plan to extend the infrastructure such that each application process uses an extra thread, to which the processes communicate with immediate shared memory communication, which closely resembles the approach of Umpire.

3.3.5 Abstractions

Tool infrastructures need to provide a mechanism that allows tool developers to interface with the components that they offer. For infrastructures that only provide a set of tool components, interfaces can suffice; whereas infrastructures with a more framework-like concept provide a higher-level abstraction that allows tool developers to plug their analyses into the infrastructure. The latter especially supports tool development with reusable components. If an abstraction allows a tool developer to provide the implementation of a tool in multiple parts, then a framework-based infrastructure can both connect the parts of the tool with its provided components and it can offer a means to connect components from multiple existing tools.

3.3.5.1 Lilith

Lilith [39, 54] provides a TBON layout that spans compute nodes or cores of a computing system with intended use cases in system monitoring and maintenance. It uses a Java object model where an API at the root of the tree allows tool developers to distribute data and Java code across the whole tree, i.e., across all compute nodes or cores of a system. The distributed code can then be executed on all nodes of the tree. The results of this execution are then gathered recursively from the leaves towards the root of the tree. Since the API at the root of the tree potentially provides access to an entire computing system, Lilith

also provides a security component that considers access rights. In summary, Lillith uses an abstraction where an API at the root of the tree allows a tool developer to utilize all leaves of the tree with a workflow that follows:

1. Distribute data and code,
2. Execute code, and then
3. Gather results.

3.3.5.2 MRNet

The tool infrastructure MRNet [129] provides a TBON layout and uses a module abstraction to provide a plugin mechanism for tool analyses. The tool developer provides so-called front-end and back-end code. The root of the TBON executes the front-end code, while the leaves execute the back-end code. For the MPI runtime verification use case, application processes represent these leaves. While MRNet does not provide instrumentation, a tool developer can create MPI wrappers that observe all MPI calls as back-end code. Both the back-end and the front-end code use an MRNet provided API to create communication streams, to write into, and to read from these streams. MRNet's API calls associate two modules with each stream: A *transformation* and a *synchronization* filter module. The transformation filter aggregates multiple events into new events, while the synchronization filter aligns incoming events for this aggregation. MRNet automatically loads both types of filters—implemented as dynamically loadable libraries—onto all nodes of the TBON, when a tool creates a new communication stream. The use of these modules and the stream abstraction allows MRNet-based tools to simplify the distribution and loading of the modules. In summary, MRNet's abstraction uses:

Front/Back-end code: Creates and uses communication streams,

Filters: Aggregate events as they progress towards the root.

Thus, MRNet provides an interface for the front-end and back-end code, while it uses a framework approach for the filter modules. The infrastructures SCI [93] and STCI [24] follow a similar abstraction and also provide a TBON layout. Like MRNet, both approaches use pluggable components to map functionality onto tool places.

3.3.5.3 PnMPI

P^{n}MPI [135] offers a module abstraction that particularly applies to tools that use in-line communication. Its abstraction represents a tool as a set of cooperating modules. As in MRNet, tool developers implement P^{n}MPI modules as dynamically loadable libraries. When a user executes an MPI application, P^{n}MPI reads a configuration file that lists all modules that apply to this run. Afterwards, P^{n}MPI loads the listed modules. The infrastructure automatically instruments all MPI calls and passes their information to all loaded modules, where it passes information according to the module order of the configuration file. With that, in a single execution, a tool user can easily combine multiple existing MPI tools. In addition, tools can provide services to each other, such that modules can cooperate.

In summary, for tools that intercept MPI calls, the module concept of P^{n}MPI along with its service concept allow for component reuse and composability. At the same time, P^{n}MPI provides no concept to spawn extra tool places and to connect it with some topology. Instead, P^{n}MPI tools either require no communication or they use inline communication.

3.3.5.4 OCM

The On-line Monitoring Interface Specification (OMIS) [107] defines an interface that allows OMIS compliant runtime (on-line) tools to share a common instrumentation[1]. The OMIS Compliant Monitoring system (OCM) [170] implements OMIS and allows runtime tools to use and extend its instrumentation. Runtime tools use the OMIS interface to specify which actions they execute for which events. Note that actions do not forward event information to the tool, but rather execute additional commands within OCM's monitoring component.

In summary, this *event-action* abstraction allows multiple tools to simultaneously use an instrumentation system, where a centralized mediator processes forwards event-action mappings from tools to application processes. This abstraction allows tool developers to only implement their tool analyses and to use OMIS to map their analyses to applicable events. OCM-G [11] extends OCM with a distributed mediator but does not detail its topology.

3.3.5.5 CBTF

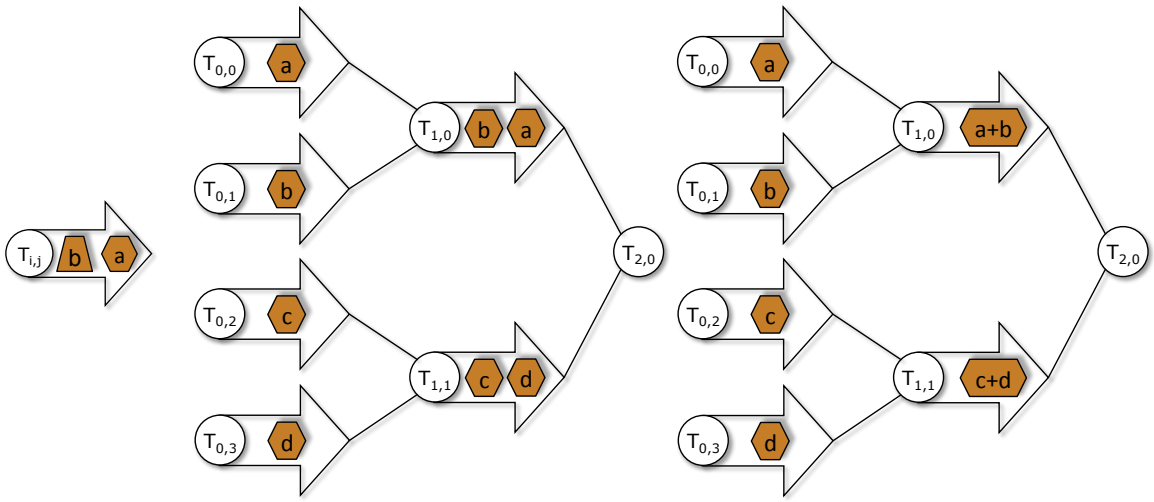
The Component-Based Tool Framework (CBTF) [49] aims at a dataflow abstraction where tool components process an input stream and create an output stream. Tool developers can connect components, where the output of one component forms the input of another. This approach is closely related to the communication graphs of DeWiz. However, a component network of CBTF can be distributed with MRNet, such that components on one hierarchy level form the input for components of the next hierarchy level. This abstraction allows tool developers to distribute tool analyses and it provides a concept for component reuse and composability. With the dataflow abstraction, CBTF components define a single interface that both serves for tool communication and for tool composability. An early experience and development report of CBTF [50] highlights that this approach of a single interface for both communication and composability poses disadvantages, since it mentions a specialized *service* component type. This component type serves for utilities that *don't fit into the data-flow model of the CBTF component framework*. However, the development of CBTF largely occurred parallel or even after the tool infrastructure developments that this thesis proposes. A first prototype of CBTF became available in August 2013, while the first releases of the tool infrastructure and correctness tools that this thesis proposes became available in November 2011.

3.4 Tree-Based Overlay Networks

While a general distributed layout of tool places—such as in Figure 3.1(b)—allows the distribution of tool analyses; some analyses map well to a hierarchical abstraction, which motivates the use of Tree-Based Overlay Networks (TBONs). Such hierarchical analyses can efficiently reduce the number of events in the network. For MPI runtime verification, events usually represent information from issued MPI calls that occur on the application processes. The root of the TBON then analyzes global properties like freedom of deadlock and operates on information from these application events. As a result, this thesis considers the flow of events from leaf nodes—application processes—towards the root of the TBON as the primary communication direction. In the following, *place* refers to a node of a TBON topology, which is either an application process—leaf place—or a tool place (node in a non-leaf hierarchy layer). Following this naming, the *root place* is the tool place that forms the root of the TBON.

If the root place receives all events from all leaf places, it would form a bottleneck. Thus, the number of events that progress towards the root must be limited. This document uses the scheme in Figure 3.2(a) to illustrate event forwarding in a TBON layout. Nodes in the illustration represent places in the TBON and use labels of the form $T_{i,j}$ where i identifies the hierarchy layer and j the index of the place within

[1] OMIS considers online monitoring that does not provide instrumentation only, but may also provide interfaces to the operating system, e.g., to stop the execution of a process. However, this difference does not impact the applicability of OMIS ideas for MPI runtime verification.



(a) A representation that illustrates event forwarding in a tool topology. Place $T_{i,j}$ first forwards event a and event b afterwards.

(b) Without event aggregation, each place forwards the events that it receives. A bottleneck results if all events are being passed to the root of a TBON topology.

(c) An event aggregation replaces incoming events with new events. The example creates new events by adding up the input events, e.g., to sum up all events.

Figure 3.2: *An illustration of the aggregation concept in a TBON topology. This concept enables highly scalable tools.*

the layer. Each leaf place $T_{0,j}$ ($0 \leq j \leq p-1$) represents an application process with rank j. The block arrow in Figure 3.2(a) illustrates the events that the place forwards towards its parent in the hierarchy—here the events a and b. Events are named, where shapes distinguish event types. As an example, each event type could require different processing or form input for different tool analyses. The example situation in Figure 3.2(a) uses two event types, i.e., the event type associated with the trapezoid shape for event b and the event type associated with the hexagon shape for event a. Further, placement of events in the block arrows indicates event order, where a place forwards the rightmost event first. In the example, $T_{i,j}$ forwards event a before event b.

3.4.1 Aggregation

Figure 3.2(b) sketches four application processes that each create a single event of identical type (hexagon shape). As an example use case, assume that each event represents a natural number and that the root receives the events in order to calculate a global sum from these values. In Figure 3.2(b), places $T_{1,0}$ and $T_{1,1}$ directly forward events to the root. In that case, the number of events that the root receives linearly grows with application scale, i.e., the root becomes a bottleneck.

In Figure 3.2(c), places $T_{1,0}$ and $T_{1,1}$ apply an *event aggregation* [6] that replaces sets of incoming events with new events. For the global sum example, $T_{1,0}$ forwards the partial sum $a+b$ instead of the individual events a and b. A TBON hierarchy allows a recursive usage of this scheme, such that each place receives one event for each of its children and forwards a single event when it receives the last of these events. Thus, in the case of a global sum, each place forwards a partial sum for the sub-tree that includes its children. Use cases of event aggregations for runtime tools include:

- Summarize performance data [117], e.g., average time spent in a function;
- Aggregate stack traces to group processes with similar behavior [7]; and
- Aggregate results of debugger queries [12].

MRNet uses its two types of filter modules to separate between aggregating events and selecting events for aggregation. Its synchronization filters select the events that belong together for an aggregation. For the global sum example, a synchronization filter would select the next available event from each child of a place. An MRNet transformation filter than uses the set of events that a synchronization filter selected and applies the actual aggregation to it, which creates a single or multiple new events.

The following uses the term *fan-in* to refer to the child count of a place. If the number of leaf places increases—application scale increases—then additional tool places can keep the fan-in constant, i.e., the TBON uses additional hierarchy layers. Thus, for aggregations such as the global sum, each place can receive constant numbers of events, even if scale increases. If the processing cost for each event is also constant across scale, this enables a constant load per place across scale. Such a design provides an efficient approach for scalable tools that exhibit event analysis costs that do not increase with scale.

3.4.2 Aggregation for Runtime MPI Verification

For runtime verification of MPI applications, aggregations apply to the following tasks:

- Detection of MPI usage errors in collective operations,
- Summarizing reports for detected defects, and
- Tool management.

The correctness tool design that Chapter 5 proposes details how aggregations facilitate the correctness checks for MPI collective operations. Often, similar MPI usage errors occur on multiple processes. In that case, an aggregation can efficiently combine these reports to provide a compact and easily accessible correctness report. Finally, tool management tasks such as startup, shutdown, and application crash-handling can utilize aggregations. These use cases motivate an evaluation of TBON topologies for runtime MPI correctness tools and suggest an advantage of these topologies over the general distributed topology.

Event aggregations also influence event order, which refers to process local order here, rather than a global order [103]. Process local order requires that if a leaf place creates an event e_x before an event e_y, then all places should also process events that include information from e_x before any event that includes information from e_y. This event order influences correctness properties like "Was the MPI datatype committed before or after the communication call?". If a place aggregates e_x—and potentially further events—into a new event e'_x then it should forward e'_x before e_y (or a derived event that subsumes it). MRNet synchronization filters do not guarantee such a property unless both e_x and e_y use the same communication stream, i.e., equal transformation and synchronization filters apply to the events. Employing aggregations for MPI usage error detection requires careful consideration of event order, since correctness analyses—such as the MPI datatype example above—depend on the order. This notion discourages the use general DAG topologies, since they can introduce additional complexity for this property, due to the existence of alternative paths on which events can arrive on a place. The infrastructure CBTF underlines this notion. While CBTF uses a DAG-based dataflow paradigm, it avoids the use of a tool topology that is a DAG. Instead it uses a TBON topology and restricts the DAGs of all non-root places to fit this model.

3.5 Applicability to MPI Runtime Verification

This section first compares the previously introduced infrastructures for their applicability in an MPI runtime error detection use case. Afterwards, a sketched tool development plan with one of these infrastructures highlights possible choices for implementing such a runtime verification tool. While this development plan provides a viable approach, it also illustrates disadvantages of the available infrastructures. This property motivates an investigation into novel abstractions and methods for infrastructures.

	Instrumentation	Topology	Means of Communication	Application Crash-Handling	Abstraction	Public Availability April 2014
MRNet	×	TBON	TCP sockets	×	Front-/Backend Code and Filters	Source
SCI	×	TBON		×	Front-/Backend Code and Filters	Source
CBTF	×	TBON		×	Component Network	Source
P^{n}MPI	✓	Inline	MPI	×	Modules and Services	Source
STCI	×	TBON	Flexible	×	Front-/Backend Code and Filters	Design
OCM	✓	Mediator		×	Event-Action Mapping	Design
Lilith	×	TBON		×	Distribute, Execute, Gather	Design

Table 3.2: *A comparison of parallel tool infrastructures highlights a lack of an infrastructure that matches all requirements for the MPI runtime verification use case.*

3.5.1 Infrastructure Comparison

Table 3.2 summarizes the infrastructures from the previous section, which are MRNet, SCI, CBTF P^{n}MPI, STCI, OCM, and Lillith. It uses the services (Section 3.2) that a runtime MPI verification tool requires as metrics and adds the public availability of the infrastructure as an additional metric. The source code for the infrastructures MRNet, SCI, P^{n}MPI, and CBTF is available and allows tool developers to efficiently utilize and extend these infrastructures. Whereas, for the remaining three infrastructures only design documents and published results are available. For OCM and Lilith, sources or binaries may still be available on request, whereas for STCI no release was available in April 2014.

The table shows that no single approach satisfies all requirements for runtime MPI verification, where especially no infrastructure provides solutions to handle an application crash. Note that MRNet provides an extension that can handle failures on tool places, but not on leaf places. Also, the design of STCI considers failure handling, but specifies that data loss may occur in case of an application crash [25].

For the instrumentation system requirement, both P^{n}MPI and OCM integrate such a system. Both approaches run tool analyses locally on application processes. P^{n}MPI uses its modules to define analyses that apply to MPI events and OCM uses its event-action mappings to define tool analyses as actions. In both cases, the approaches execute tool analyses on the application processes. This setting does not support offloading-based tools that use places with their own control flows to offload some tool analyses. OCM allows that actions provide a reply to a tool, but with its mediator topology this yields a centralized topology that limits scalability. The remaining infrastructure approaches do not offer instrumentation, but offer a TBON as topology. This topology enables a distributed implementation of an offloading tool for MPI. Limited information on the communication medium, and whether it is exchangeable, is known for some of the infrastructures; where MRNet and P^{n}MPI use a fixed means of communication, while STCI's design targets exchangeability.

Finally, the abstraction both impacts how tool analyses interact with instrumentation and also influences composability and component reuse. Especially P^{n}MPI's module concept with its MPI focused instrumentation allows for such characteristics. Also OCM's event-action mapping allows tool developers to specify what tool analyses apply to which events in a flexible manner that enables tool interoperability. The remaining abstractions do not interlink with an instrumentation component and focus on a distribution of tool analyses on a TBON. MRNet, SCI, and STCI use a communication stream abstraction that applies transformation filters. CBTF, on the other hand, uses a dataflow abstraction that connects outputs of components as inputs of connected components.

3.5.2 A Tool Design with Existing TBON Infrastructures

In summary, none of the infrastructures from Table 3.2 directly provides all the services that an MPI runtime verification tool requires. The table highlights concepts that integrate an instrumentation service, as well as concepts that provide a topology for distributed and scalable event processing. However, no approach provides a combination of the two services. Additionally, no approach provides a mechanism to handle an application crash.

Also note that a direct combination of say P^{n}MPI and MRNet does not combine their advantages. That is due to the fact that P^{n}MPI provides an abstraction that considers instrumentation, but an extension would need to specify how its instrumentation interacts with tool places on MRNet's TBON topology.

The following describes a coarse-grain design for a scalable MPI runtime verification tool that is based on these existing infrastructures. Since scalability is a primary goal of this thesis, a selection from these existing infrastructures should favor a topology that supports scalability over an instrumentation system. As a result, MRNet, SCI, CBTF, STCI, and Lilith would qualify. However, Lilith has its primary uses cases in system monitoring rather than in runtime tools, and thus, is less applicable for the target tool. Of the remaining infrastructures MRNet would be the most natural choice, since it demonstrated applicability to multiple use cases—e.g., TAUoverMRNet [117], STAT [7], and LaunchMON [1]—and operates at scales of 200,000 processes [104] and beyond. None of the other infrastructures demonstrated either of these properties, but they closely match the capabilities of MRNet. CBTF provides an extension to MRNet if a detailed tool design for the target tool would identify an advantage for the dataflow abstraction.

Thus, an MRNet-based tool design—or one with a similar infrastructure—could provide the services that an MPI runtime verification tool requires as follows:

- MRNet provides a TBON topology,
- The wrapper generator in mpiP [158] could create an instrumentation system that forms the back-end code for MRNet,
- Handwritten code forms the front-end code that writes the global correctness report and checks for global MPI usage errors,
- MRNet filter modules would apply event aggregation where possible to ensure that the front-end does not become a bottleneck, and
- An extension of Umpire's application crash-handling scheme to TBONs and MRNet's components provides a crash handling scheme.

3.5.3 Discussion

The above sketch of a design could yield a detailed design that enables the development of a scalable runtime MPI verification tool. However, from a tool developer perspective it comes with several drawbacks:

I: The design employs an infrastructure along with additional components for instrumentation, thus, a tool developer must both learn the handling of the infrastructure and of the instrumentation components; An integration of an infrastructure with an instrumentation system (e.g., such as in OCM) would be desirable;

II: Tool analyses are spread over the front-end, the back-end, and the filter code; Each of these codes uses a separate interface; Thus, restricting the composability and reuse of the analyses, as in the initial study of Marmot and Umpire (Section 3.2);

III: MRNet only considers event order within communication streams, but an MPI correctness tool requires information on the order of events to which different aggregations could apply;

IV: Pure TBON topologies have inefficiencies for scalable analysis of MPI point-to-point operations as subsequent sections illustrate.

The first drawback (I) is more of a development inconvenience that requires the use of further tool components, besides the parallel tools infrastructure. However, drawback II raises a serious development concern. A situation where tool analyses are split over front-end, back-end, and filter code (each with their specific interfaces) directly limits the reuse and composability of these analyses. If a developer decides to implement a certain analysis directly on the back-end code (on the application processes), then another developer that wants to reuse this analysis can't easily migrate this analysis to say the first tool hierarchy layer, e.g., in order to reduce application perturbation. This situation closely resembles the situation around the correctness checks of Marmot and Umpire. Clearly, a well-designed implementation of the tool analyses could provide bridging interfaces to overcome such limitations, but it requires developer awareness of the issue and good development practices. A key question from a tool developer perspective is:

> *Can an infrastructure provide a single concept to put tool analyses flexibly onto the application processes, the root of the TBON, or intermediate hierarchy layers?*

Drawback III applies to MRNet's communication stream concept. Each stream applies a specific event aggregation, e.g., specific transformation filters, and uses synchronization filters to define an event order within the stream. If a tool analysis applies to events that use different event aggregations, i.e., that use different streams, then MRNet provides no order between these events. For some analyses of the MPI correctness use case, event order is an important criterion. In such cases, tool developers must provide their own solutions, one of which is timestamping events. As subsequent sections illustrate, these timestamps introduce scalability problems in the presence of event aggregations. Thus, a further question from a developer perspective is:

> *Can an infrastructure automatically preserve event order in the presence of event aggregations?*

The last drawback (IV) impacts correctness analyses for MPI point-to-point operations. Subsequent sections illustrate that their analysis in a TBON topology can limit the efficiency of event aggregations that apply to these events. For such cases, the distributed topology in Figure 3.1(b) provides advantages, since it enables a direct communication between all pairs of tool places. An extension of a TBON infrastructure that incorporates a communication system with a higher connectivity would provide scalability improvements for these correctness analyses.

The questions that drawbacks II and III raise, directly apply to MRNet's—and also STCI's, SCI's, and CBTF's—abstraction. A pure incremental approach won't overcome these drawbacks, since both directly result from the concepts of front-end/back-end/filter code and the communication streams. This situation motivates an exploration into novel concepts for parallel tool infrastructures that overcomes the above drawbacks, rather than a tool design that uses existing infrastructures. Such an investigation can address application crash-handling and flexibility for the means of communication at the same time, since MRNet also imposes limitations for both of these properties.

4 New Methods for Parallel Tools Infrastructures

> *For a maximum flexibility, the monitoring interface should provide orthogonal sets of event and action services and should allow tools to freely associate actions with events occurrences.* [169]

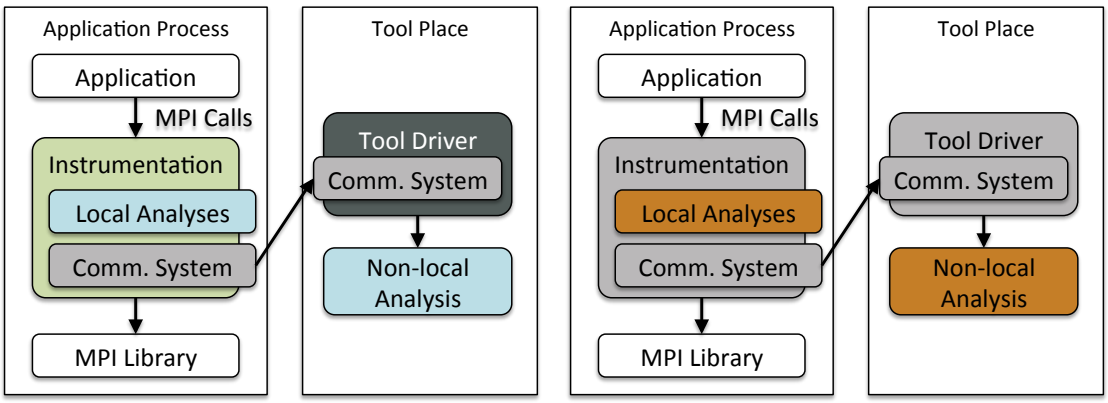
Based on the conclusions of the previous chapter, this chapter summarizes the design and concepts for a new parallel tools infrastructure, whose prototype implementation is named the Generic Tools Infrastructure (GTI). Its focus is to meet all requirements for MPI runtime verification and to address the drawbacks that arise around a development approach with current TBON abstractions. At the same time, the parallel infrastructure should apply to use cases beyond MPI runtime verification. Section 4.1 summarizes the abstraction for GTI and relates it to existing infrastructures. Afterwards, Section 4.2 details the key concepts of this abstraction. Based on these basic concepts, Section 4.3 presents advanced concepts in GTI, which especially incorporate the event aggregations that provide scalability for TBON-based tools. Section 4.4 then details key algorithms that enable the instantiation workflow on which GTI builds. Methods to preserve process local event order in the presence of event aggregations provide a further requirement for MPI runtime verification uses cases (Section 4.5). A comparison relates this document to previous publications on GTI and its abstraction (Section 4.6). Finally, Section 4.7 compares the requirements for MPI runtime verification with the capabilities of the GTI prototype, which implements the methods that this chapter introduces.

4.1 Abstraction

The subsequent abstraction targets a tool development where *the tool developer can focus on the development of the tool analyses*. Figure 4.1(a) repeats the previous summarization of the components that an offloading-based runtime tool requires (from Figure 2.11). Of these components, the subsequent abstraction proposes a tool infrastructure that provides instrumentation, a communication system, a tool topology, and tool place drivers, i.e., the components with gray background in Figure 4.1(b). The tool developer then only provides the local and non-local analyses.

The parallel tool infrastructure abstraction that this thesis proposes uses the tool analyses as central elements. An analysis can flexibly execute on the application processes or a tool place. Depending on where an analysis executes, the infrastructure takes care to provide all input that the analysis requires, irrespective whether it is interested in information that occurs remotely. The following illustrative example highlights some of the options that an analysis-centric tool development enables:

1. The developer of an MPI correctness tool designs and implements a correctness check as an analysis.
2. The developer's initial tool layouts put this analysis directly onto the application processes.
3. A latter performance investigation reveals that the analysis slows down the application, due to heavyweight execution time.
4. Performance optimized tool layouts could then use additional tool places and execute the same analysis there instead, as to remove the analysis from the application's critical path.
5. No modifications would be necessary to adapt the analysis for the new layouts and the infrastructure would automatically manage tool internal communication to forward information from application processes to the tool places that now run the analysis.



(a) An offloading-based tool requires the illustrated tool components (from Figure 2.11).

(b) The thesis proposes: Tool developers only provide the tool analyses (boxes with labels "local analyses" and "non-local analyses"). The tool infrastructure then provides all remaining components (boxes with gray background).

Figure 4.1: *An illustration with tool components highlights the overall tool infrastructure proposal of this thesis.*

This illustration uses tool places, for which subsequent sections consider a wide range of layouts that include TBONs, but retain the idea of hierarchy layers. The above illustration clearly highlights an important characteristic:

> *If the infrastructure shall forward input to analysis automatically, then it needs knowledge on what information each analysis requires.*

Specifications serve this purpose and describe which events—occurrences of functions or communication calls as examples—should trigger which analyses. As an example, a specific correctness check might be triggered when `MPI_Send` is issued by the application. Thus, specifications describe a relation between events of interest and the analysis that they should trigger. While it allows a tool infrastructure to forward all relevant information to analyses, it also allows the infrastructure to create and control the instrumentation system itself. Further, the above notion of *the infrastructure triggers the analyses* immediately highlights the presence of a framework control flow. Thus, the overall control flow is with the infrastructure and it only triggers analyses when information on their input events arrives.

The final tool infrastructure element that is missing is parallelism. The infrastructure operates for a parallel application and should enable distributed analyses for scalability. An *event-flow* definition serves this purpose and determines which analysis on the same or other places will be triggered when an event occurs on a certain place. Thus, an event on one place can trigger analyses on remote places. This concept enables the specification of distributed analyses that operate outside the control flow of the application.

The remainder of this section details an abstraction with the five ideas above:

- Analysis-centric development,
- A tool layout with application processes and additional tool places,
- Specifications that describe events as well as the relation between events and analyses,
- A framework like control flow triggers analyses, and
- An event-flow definition that describes which analyses will be triggered (on the current or on a remote place) when an event occurs.

A terminology introduction highlights key terms of the abstraction first. An illustration of a minimal tool instance then elaborates how these concepts work together. Subsequent sections then formally introduce these concepts, as well as algorithms that operate on these definitions. Particularly this includes a specification of the *event-flow* that defines the workings of the proposed abstraction.

4.1.1 Terms

Key terms in the proposed abstraction are:

Analysis: A [tool] analysis represents a part of the tool functionality,

Module: A module implements a set of analyses,

Hook: A source of events that the infrastructure can instrument,

Analysis-Hook-Mapping: This mapping specifies the input relationship between hooks and analyses,

Operation: Operations transform arguments of hooks and provide derived/preprocessed inputs to analyses,

Layer: Layers are sets of places and define a hierarchy that creates the overall tool layout,

Place: A place is either an application process/thread (application place) or a tool owned process/thread (tool place),

Layer-Module-Mapping: This mapping specifies which modules run on the places of what layers,

Event: An event results if a place triggers an instrumented hook, and

Event-Flow: Each hook has a communication direction that impacts which analyses of what places will be informed about the event.

Analyses implement the tool functionality and *modules* contain sets of these analyses. Modules serve as a packaging concept to group analyses of one type together and to provide them access to common data in the GTI implementation. As an example, in MPI runtime verification, each check that applies to MPI datatype validity could be one analysis, and be packaged into a module for datatype related checks. This module could hold a list of user-defined MPI datatypes, to which all analyses of the module have access. Further, modules can have functional dependencies to other modules. As an example, the module for MPI datatype related checks could alternatively have a dependency to a module that tracks all user-defined datatypes. This allows modules to use services provided by other modules.

Hooks describe what the instrumentation system can observe. The GTI implementation considers functions of the programming language C as the primary instrumentation type. Future extensions could consider other mechanisms. The *analysis-hook* mappings then connect hooks and analyses to describe which arguments of a hook form the input for an analysis. This implies that the mapped analysis should be informed whenever the hook is issued (subsequent conditions detail this concept). The *operations* then provide argument transformations to preprocess inputs of analyses, e.g., to calculate a sum from an array.

The *layers* specify the topology of the tool. Each layer consists of a set of places and represents one hierarchy level of a tool topology. The *layer-module* mapping then maps modules onto layers of the hierarchy. This both defines which places execute what modules—along with their analyses—and it impacts how information of triggered hooks is forwarded within the tool. If a hook gets triggered, GTI's instrumentation system creates an *event* and communicates it to all places that require information on this event. The *event-flow* determines which places require such information. The primary communication direction is from the application processes to the highest hierarchy layer, e.g., a root of a TBON layout. Simplified, event-flow for the primary communication direction specifies that if a hook h creates an

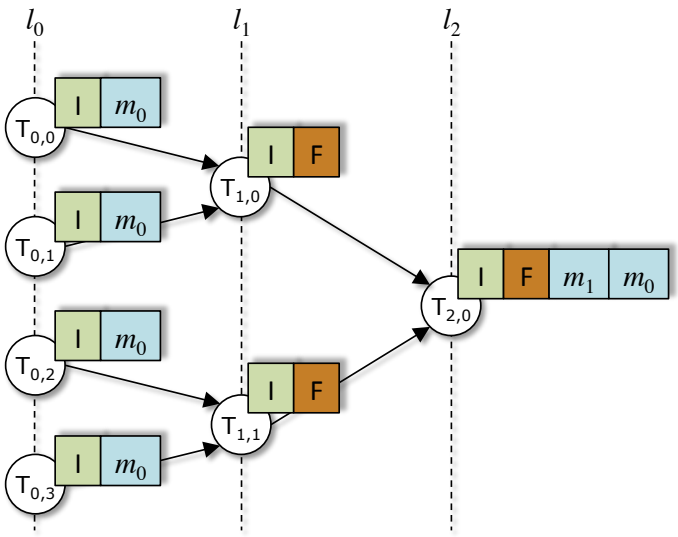


Figure 4.2: *The proposed abstraction employs a tool layout with layers of places and a flexible mapping of tool analyses onto these layers. An infrastructure can provide tool components such as instrumentation ("I") and event forwarding ("F"), while the tool developer provided the actual analyses (m_0 and m_1). A tool specification, such as sketched in Figure 4.3, allows an implementation of the proposed abstraction to instantiate the tool in this figure.*

event on a place, then all connected places on higher hierarchy layers that execute a module m with an analysis-hook mapping to h will receive information on this event.

An infrastructure that implements an abstraction with these terms requires the tool developer provided implementation of the analyses and modules, as well as a description that connects them with hooks and a tool layout. The GTI prototype employs a specification language to let tool developers provide this description. The term *specification* refers to a description in this specification language. As an example, a tool specification would provide a specification for all hooks of interest, as well as for the mapping between analyses and hooks. An instantiation system then combines the implementations of the analyses with their specifications to generate instrumentation and tool internal communication services that match the needs of the described tool.

4.1.2 Layout and Analyses

Figure 4.2 highlights the introduced terms with an example tool instantiation that uses three layers of places. The first layer, l_0, consists of the four application places $T_{0,0}$–$T_{0,3}$ that could represent four MPI processes with ranks 0–3, the second layer l_1 consists of places $T_{1,0}$ and $T_{1,1}$, and the third layer l_3 of the single place $T_{2,0}$. The figure uses similar representations as the previous tool architecture diagrams, e.g., Figure 2.10 on page 16. The boxes with label "I" in Figure 4.2 again highlight instrumentation components, i.e., components that observe sets of hooks. The additional boxes with label "F" highlight components that implement event forwarding in the tool (subsequent sections detail the role of these components). As opposed to a development with custom components, both the instrumentation and forwarding component result from an instantiation system that processes the tool specification. Finally, the blue boxes m_0 and m_1 represent modules that are mapped onto layers. The layer-module mappings always apply a mapped module to all places of this layer, rather than just specific places. The following uses the term *module instance* to refer to a specific instance of a module, e.g., in Figure 4.2, m_0 on $T_{0,2}$ is a module instance. In summary, a tool instance that results from a tool specification closely resembles an offloading-based tool with a topology of tool places, but all components except for the actual analyses (packaged in modules) are provided by the infrastructure.

The tool instance in Figure 4.2 allows a more detailed definition of the event-flow: Tool instances forward events along the primary communication direction, unless a hook specifies that is uses another communication direction. The arrowheads in Figure 4.2 illustrate the primary communication direction, which is from application places towards higher-level layers. To illustrate the event-flow definitions informally, let a acyclic directed graph G represent the topology of tool places with its primary communication direction. An event e results if a hook h of the instrumentation system on a place T is triggered. The event contains information—arguments or operation results—from h. First, place T triggers all analyses of modules that are mapped onto T and that have an analysis-hook mapping to h. Afterwards, place T forwards e to a child T' in the topology graph G, if T' or a successor of T' executes a module m that contains an analysis that has an analysis-hook mapping for h.

The following examples illustrate this definition with the tool instance in Figure 4.2: If m_1 has an analysis-hook mapping to a hook h, then the application places $T_{0,0}$–$T_{0,3}$, as well as the tool places $T_{1,0}$, $T_{1,1}$, and $T_{2,0}$ would forward events that result from h to the module instance of m_1 on $T_{2,0}$. At the same time, the module instance of m_0 on application place $T_{0,0}$ is only triggered for events that occur on this place itself. Particularly, place $T_{1,0}$ would not forward events for h to $T_{0,0}$, since the latter place is neither a child nor a successor of the former place. In summary, modules on a hierarchy layer receive events from all lower-level layers (for hooks to which an analysis of the module is mapped).

Subsequent sections introduce event aggregations and filters that reduce the number of events that progress towards higher-level hierarchy layers. Modules that are marked as aggregations serve for this purpose. The user does not map these modules manually, but rather the tool infrastructure automatically determines whether an *aggregation module* is applicable for a specific layout.

4.1.3 Tool Specification

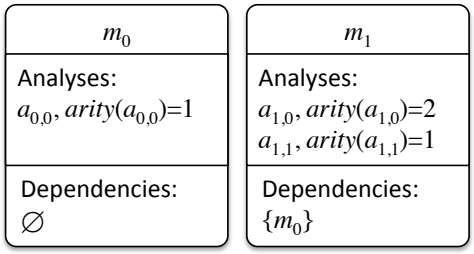
An instantiation system reads specifications for modules, analyses, hooks, operations, analysis-hook mappings, layers, and layer-module mappings to instantiate a tool with the proposed abstraction. To illustrate the inputs of this process, consider the tool instance from Figure 4.2. For this instance, Figure 4.3 sketches the input specifications for the instantiation system. It illustrates the specifications for two example modules (Figure 4.3(c)), two hooks (Figure 4.3(a)), one operation (Figure 4.3(b)), analysis-hook mappings (Figure 4.3(d)), three layers with their connections (Figure 4.3(e)), and layer-module mappings (Figure 4.3(f)).

The two modules m_0 and m_1 in Figure 4.3(c) contain the analyses $a_{0,0}$, as well as $a_{1,0}$ and $a_{1,1}$ respectively. The illustrated function arity specifies the number of arguments for analyses, operations, and hooks. As an example, the analysis $a_{1,0}$ of m_1 expects two arguments as input. In addition, the module m_1 has a functional dependency to module m_0, i.e., it relies on services of m_0. As an example for such a dependency, recall the example of a module for MPI datatype related checks (as m_1) and a module that tracks user-defined datatypes (as m_0).

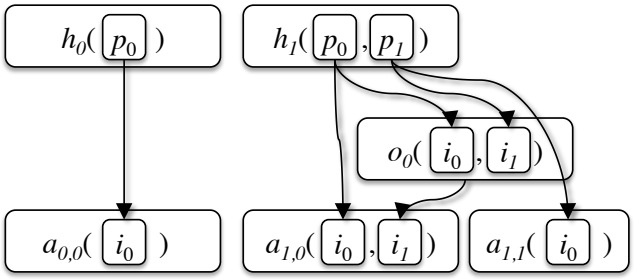
Figures 4.3(a) and 4.3(b) define two hooks (h_0 and h_1) and one operation (o_0). Based on these analyses, hooks, and the operation, Figure 4.3(d) sketches analysis-hook mappings with arcs that represent an *is-input-for* relationship. A later section details a definition for such argument mappings. The example maps the analysis $a_{0,0}$ to hook h_0 and specifies that the first argument of h_0 provides the single argument that $a_{0,0}$ expects. The mapping of $a_{1,0}$ uses the operation o_0 to compute the second input argument of $a_{1,0}$. For that, a mapping of the operation o_0 onto h_1 specifies the inputs for the operation in turn. The formalizations in subsequent sections consider operations that use hook arguments as their inputs and that return a single argument (either a scalar or an array). The analysis-hook mappings both enable a selection of hook arguments for analysis inputs and to transform or preprocess these arguments with operations. Assume that $a_{0,0}$ represents a correctness analysis for integer values, which checks that its input value i_0 satisfies $i_0 \geq 0$, such as to detect defects like the negative count that can occur in the incorrect MPI usage scenario of Figure 2.5 (page 9). If the hook h_0 represents a simplified version of the MPI call `MPI_Send` that only receives a *count* value as its input, then the mapping of $a_{0,0}$ to h_0 would provide the *count* argument of `MPI_Send` to the correctness analysis $a_{0,0}$.

(a) Specifications of available hooks h_0 and h_1 describe the sources for events.

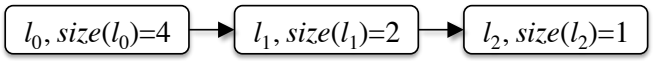
(b) A specification of an operation o_0 allows a tool to apply transformations to the arguments of a hook.



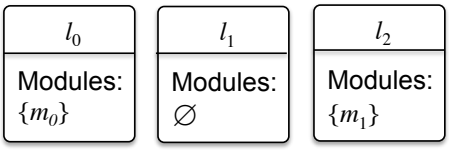
(c) Specifications for tool-developer-provided modules m_0 and m_1, along with their analyses $a_{0,0}$, $a_{1,0}$, and $a_{1,1}$, describe the available analyses to the tool infrastructure.



(d) An example specification of an analysis-hook mapping details which analyses apply to which hooks, along with details for the hook and analysis argument mappings.



(e) A layer tree with a root layer l_0 (application) and two tool layers l_1 and l_2 illustrates a specification for a tool layout.



(f) A specification for the layer-module mapping details which modules (along with their analyses) execute on which layers.

Figure 4.3: *The use of tool specifications, as illustrated in this figure, enables the proposed analysis-centric tool development approach. The illustrated specification yields a tool layout and analysis placement as in Figure 4.2.*

Figure 4.3(e) sketches the specification of the three tool layers l_0, l_1, and l_2. The first layer, l_0, represents the application places, while l_1 and l_2 represent layers of tool places. The function $\text{size}(l)$ returns the amount of places for a layer l, e.g., the example uses four application processes ($\text{size}(l_0) = 4$). In addition, the arcs between the layers in Figure 4.3(e) specify the primary communication direction. As an example, the connection (l_0, l_1) represents that l_0 forwards events towards l_1. Based on the specifications of the layers, Figure 4.3 illustrates the layer-module mapping, which maps module m_0 onto layer l_0 and module m_1 onto l_2. The instantiation system must evaluate module dependencies when it generates a tool instance. The module dependency of m_1 to m_0 requires that the instantiation system also maps m_0 onto places that use m_1, as the illustration in Figure 4.2 highlights.

The following sections detail these specifications and define the process that creates a tool instance from them.

4.1.4 Relation and Comparison to State-of-the-Art

The proposed abstraction closely relates to concepts from P^{n}MPI [135], MRNet [129], and OCM [170].

Similarly to the proposed abstraction, both MRNet and P^{n}MPI use modules. They serve for composability and to increase the reusability of tool components. While modules in MRNet have neither functional dependencies nor a service concept to cooperate with each other, modules in P^{n}MPI can provide services to each other. Modules in the proposed abstraction can specify dependencies between each other instead. This enables them to share services with each other and it ensures that if a module requires services from another module, then the tool infrastructure is able to ensure that dependent modules are present where they are required. As a result, module dependencies strengthen the service approach of P^{n}MPI [136], since they provide a clear specification of required dependencies that ensures correct op-

eration of tools that rely on module cooperation.

The proposed abstraction relates closely to MRNet in two aspects: First, it triggers analysis with a framework approach, which resembles how MRNet triggers filter modules; Second, it uses a topology of tool places to enable distributed and scalable event analysis. The topologies, with layers of places, provide TBON layouts as well as other types of layouts, as subsequent sections illustrate. In MRNet, filter modules get triggered when events occur on their associated communication streams. However, MRNet describes the events on the communication streams and their relation to the filter modules in its front-end and back-end code. The proposed abstraction uses specifications as this means instead. As a result, the proposed abstraction can apply modules on all hierarchy layers of the tool, including application processes, as well as the root of the hierarchy. In comparison to MRNet this ensures that all analyses can be mapped freely onto all places, rather than just specific places. Thus, it completely avoids the need for tool developer provided back-end and front-end code. Additionally, the proposed abstraction avoids the use of communication streams, which limit event order in MRNet. The event-flow definition replaces the concept of communication streams. Subsequent sections highlight how an event order preserving aggregation scheme enables an event order between all events, rather than just between similar events on the same stream.

A further difference to MRNet is that the proposed abstraction uses a tool layout specification in the form of layers, layer connections, and layer-module mappings. This allows an instantiation system to generate tools that are specifically adapted to a certain use case. The flexibility includes the specification of the tool topology and of the modules that execute on the topology. As an example, if for the MPI runtime correctness use case, a tool user is only interested in investigating deadlocks then he can configure the tool to only use modules that contain analyses for deadlock detection, rather than any other correctness checks. MRNet on the other hand provides flexible TBON topologies (number of layers and number of places for each layer), but the selection of filter modules is coded into the front-end and back-end code. As a result, this code would need to provide the necessary flexibility to only select certain tool analyses, as well as the dependency tracking that ensures that such a module subset retains all required functionality.

Finally, OCM's event-action mappings closely relate to the analysis-hook mappings in the proposed abstraction. However, in OCM, actions execute on application threads or processes and only provide a reply to the tool. OCM does not consider distributed tools that use additional tool places to distribute their analysis on more than just the application processes. The proposed abstraction carefully extends event-action mappings onto distributed tool layouts and uses the event-flow to define how events need to be forwarded in this topology.

4.2 Formalization and GTI

After a short introduction of notations, this section formally defines the terms from the last section to detail the concepts of the proposed abstraction. Subsequent sections then introduce advanced concepts such as event aggregations and use the formalization here to provide key algorithms of the instantiation system. Besides a formalization of the terms of the abstractions, this section summarizes the GTI prototype implementation and highlights how it realizes individual concepts. Especially, this impacts choices for implementing modules and for providing the tool internal communication system.

The following first formalizes the module term and then details how GTI implements them. Afterwards, this section introduces the concept of *layer trees* that allows tool developers to specify the layout of a tool. Connected nodes in a tool topology must communicate, where GTI uses two types of communication modules to provide a flexible communication system. A formalization of the hook and operation terms then enables a definition of the analysis-hook mappings. These definitions allow a clear specification of the event-flow that defines the activities of a tool with the proposed abstraction. Finally, this section summarizes how GTI allows tool developers to provide the specifications for analyses, modules, hooks, operations, and their mappings.

The symbol list for this thesis on page 161 summarizes all formalizations of this and the subsequent

sections and serves as a notation reference.

4.2.1 Notations

The following uses $\mathbb{N}$ as the set of natural numbers and assumes $0 \in \mathbb{N}$. Further, $\mathcal{P}(A)$ is the powerset of a set A, which is the set of all subsets of A. Finally, the following sections use $\{\bot, \top\}$ to define a Boolean domain ($\{\text{false}, \text{true}\}$) and use $\wedge$ as the logical AND operator, as well as $\vee$ as the logical OR operator.

This document follows common definitions of graphs and directed graphs. Particularly, a directed graph $G = (N, E)$ consists of a set of nodes N and a set of edges $E \subseteq N \times N$. An arc $(n, n') \in E$ is directed from node n to node n'. The fan-in of a node $n \in N$ is $|\{n' : (n', n) \in E\}|$ and in analogy the fan-out of n is $|\{n' : (n, n') \in E\}|$. The functions $\text{fanin}(n, G)$ and $\text{fanout}(n, G)$ return these values for a node n of a directed graph G. Additionally, a sink $n \in N$ is a node with $\nexists n' \in N : (n, n') \in E$. The function $\text{successors}(n, G)$ returns the set of successor nodes of node $n \in N$ in the directed graph $G = (N, E)$ ($\text{successors}(n, G)$ is the smallest set that includes $\{n' | (n, n') \in E\}$ and that satisfies: if n' is in the set then all n'' with $(n', n'') \in E$ are also in the set). Similarly the function $\text{predecessors}(n, G)$ returns the set of predecessor nodes of node n.

Also the use of trees follows common definitions. Particularly, rooted, directed trees $T = (N, E, r)$ are used in subsequent sections, which are defined as: (N, E) being a directed graph that is connected, cycle free, and $r \in N$ being the root of the tree (e.g., see [59]). Further, all arcs are either directed towards or away from the root, depending on the use case of the tree. In the latter case, the following sections call such a tree an arborescence (e.g., see [45]).

4.2.2 Modules

The following details how GTI generalizes the introduced module term to implement wide ranges of its functionality with a hierarchy of cooperating modules. Afterwards, a formal definition underlines the module and analysis terms of the proposed abstraction, along with the concept of module dependencies.

4.2.2.1 GTI Modules

The proposed abstraction uses modules to package multiple analyses, which operate on the same data, together. GTI uses dynamically loadable libraries to implement these modules. This follows the implementations of P^{n}MPI as well as MRNet that also both use dynamically loadable libraries to implement their modules. Further, GTI also represents its own implementation—drivers for places, communication implementation, generated instrumentation, and event forwarding—as modules. Thus, a GTI tool instance is a collection of modules that cooperate to implement a tool. The actual modules that implement analyses are consequentially a subset of this collection.

GTI uses functionality of P^{n}MPI to load and initialize its modules. Particularly, GTI uses a regular installation of P^{n}MPI as a base service to reuse module management capabilities, as well as MPI instrumentation capabilities. A GTI module can have multiple instances, e.g., one module that implements tool internal communication capabilities could have two instances on a tool place, one that serves for communicating towards higher hierarchy layers and one that serves for communicating towards lower hierarchy layers. As opposed to P^{n}MPI, where each module only has a single instance, GTI uses object instances of C++ classes to represent module instances and extends the P^{n}MPI module concept with a notion of instances. A DAG (Directed Acyclic Graph) then organizes all module instances at runtime. A parent module retrieves pointers to interfaces of all of its child modules during instantiation. Thus, each module instance gains access to the interfaces of its child modules, which serves as the concept for module cooperation. Most importantly, this concept implements the dependencies that analysis modules can have in the proposed abstraction. A module instance with a dependency gains a child pointer to the interface of the dependent module instance. Thus, depending modules can use services provided by dependent modules.

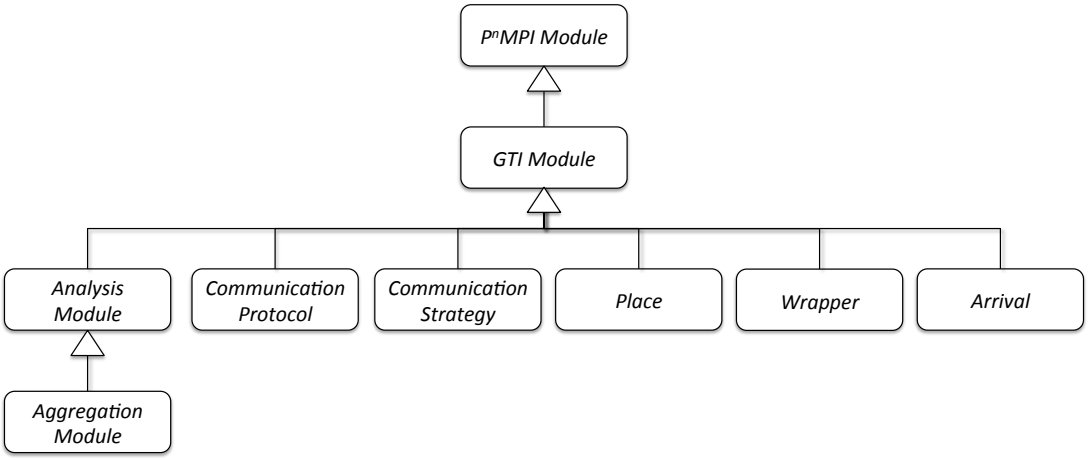


Figure 4.4: *The GTI prototype uses "Analysis Modules" to represent tool developer provided modules of the proposed abstraction, while it also uses further types of modules for its own implementation.*

Figure 4.4 summarizes the module types in GTI with class diagram semantics. The *analysis modules* represent the modules of the proposed tool infrastructure abstraction. Tool developers provide them. A specialized version of an analysis module is an *aggregation module* that serves for event aggregation. *Communication protocol* as well as *communication strategy* modules are part of GTI's communication system, *place* modules implement the main loop for tool places, and *wrapper* as well as *arrival* modules result from GTI's instantiation workflow. Subsequent sections detail each of these module types.

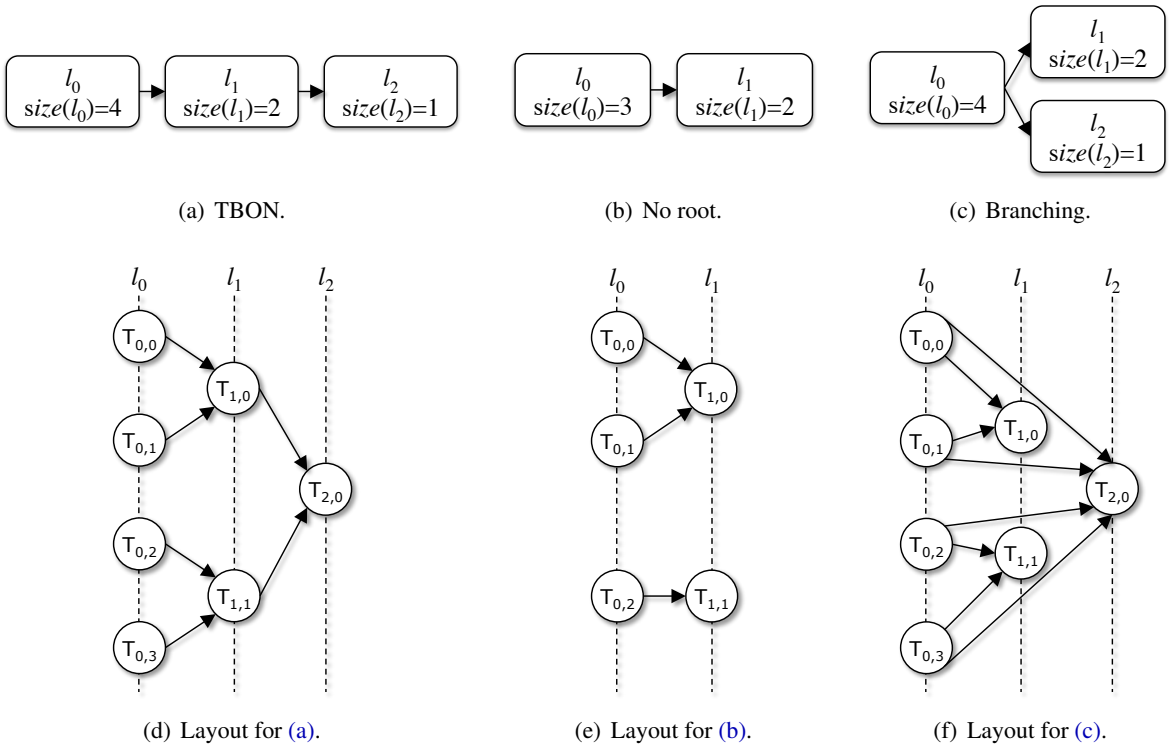
4.2.2.2 Module Term of the Abstraction

To formalize the specifications that describe modules, let A be the set of tool analyses, where the function $\text{arity} : A \to \mathbb{N}$ specifies the number of arguments that each analysis expects. Let $M = M_T \cup M_A$ be the set of modules where M_T (tool analyses) is the set of analysis modules and M_A the set of specialized aggregation modules (subsequent sections). Further, M_T and M_A must be disjoint. These two types of modules must be discerned for the proposed abstraction, since tool developers can only map analysis modules onto the tool topology, whereas the infrastructure automatically maps aggregation modules were applicable.

Each module implements a set of analyses, where the function $a_M : M \to \mathcal{P}(A)$ associates a set of analyses with each module. Let a restriction apply to this mapping: each analysis may only be used by one module, i.e., $\forall m, m' \in M$ with $m \neq m'$ holds $a_M(m) \cap a_M(m') = \emptyset$. This restriction only serves to avoid the use of redundant tool functionality, which could result from mapping multiple modules with similar analyses. Rather, module dependencies support scenarios where multiple modules require similar functionality. In that case one module m implements the analysis, while other modules m' depend upon m.

The function $d_M : M \to \mathcal{P}(M_T)$ defines module dependencies. Modules may only depend on analysis modules, since aggregation modules will be automatically mapped in the abstraction. GTI's implementation organizes module instances in a DAG. Thus, this implementation supports soft-dependencies since d_M may impose circular dependencies that would not map to such a hierarchy. The implementation ignores soft dependencies for creating the hierarchy, but still provides access to module interfaces for them.

To illustrate these definitions, consider the scenario in the introductory example of Figure 4.3(c). The following provides formal definitions for this scenario: $A = \{a_{0,0}, a_{1,0}, a_{1,1}\}$, $M_T = \{m_0, m_1\}$, $M_A = \emptyset$, $a_M(m_0) = \{a_{0,0}\}$, $a_M(m_1) = \{a_{1,0}, a_{1,1}\}$, $d_M(m_0) = \emptyset$, and $d_M(m_1) = \{m_0\}$.



(a) TBON. (b) No root. (c) Branching.

(d) Layout for (a). (e) Layout for (b). (f) Layout for (c).

Figure 4.5: *An illustration of three layer trees (a)-(c) and their resulting tool topology graphs (d)-(f) highlights the types of topologies that the proposed tool infrastructure abstraction considers.*

4.2.3 Topology

The proposed abstraction uses layers—containing places—to define tool layouts. The concept of layers then serves to simplify the association between places and their analyses, as well as to specify the means of communication between places. The following definitions formalize this concept. Let $L = \{l_0, l_1, \ldots, l_n\}$ be the set of tool layers, where l_0 represents the layer of application places. The remaining layers represent tool places. A so-called *layer tree* represents the layer connections as a directed, rooted tree $\mathcal{L} = (L, E_\mathcal{L}, l_0)$ with root l_0[1]. Or more concisely, a layer tree is an arborescence with root l_0. Particularly, this implies that there exists exactly one path from l_0 to each layer $l \in L \setminus \{l_0\}$. The layer tree ensures that each layer can receive events from the application places in l_0 and it ensures that only one such path exists, which simplifies subsequent considerations of event order. The function $\text{size} : L \to \mathbb{N} \setminus \{0\}$ specifies the number of places for each layer. The layer tree must satisfy $\text{size}(l) \geq \text{size}(l')$ if $(l, l') \in E_\mathcal{L}$. Thus, layer size stays equal or decreases for higher hierarchy layers, which reflects the notion that higher-level hierarchy layers reduce or condense data along the primary communication direction (from the application places in l_0 towards higher hierarchy layers).

Figures 4.5(a)–4.5(c) illustrate three layer trees and their layer sizes (result of the size function). The layer tree in Figure 4.5(a) represents the layout of the introductory example from Figure 4.3, i.e., $L = \{l_0, l_1, l_2\}$, $\mathcal{L} = (L, E_\mathcal{L}, l_0)$ with $E_\mathcal{L} = \{(l_0, l_1), (l_1, l_2)\}$, and $\text{size}(l_0) = 4$, $\text{size}(l_1) = 2$, as well as $size(l_2) = 1$.

Based on layer trees, the proposed abstraction creates a *tool topology graph* as a directed graph $\mathcal{T} = (N, E_\mathcal{T})$. For each layer $l_i \in L$, the node set N of the tool topology graph includes nodes $T_{i,j}$ with $j \in \{0, 1, \ldots, \text{size}(l_i) - 1\}$ for the places. Based on these nodes for the places of the individual layers, a connection rule specifies which places are connected. For the connection rule, two places are connected if the layers to which they belong are connected in the layer tree, e.g., $T_{i_1,*}$ and $T_{i_2,*}$ could be connected

[1] Note that l_0 is a *root layer* of the layer tree, but not the root of a topology of places, e.g., not a TBON root.

if $(l_{i_1}, l_{i_2}) \in E_{\mathcal{L}}$. Further, the connection rule uses the second index of the nodes for the places to specify which places can be connected. Different rules can follow different goals in balancing the connections between the places of connected layers. The default topology graph construction that the GTI prototype employs is:

- $N = \{T_{i,j} : l_i \in L \wedge 0 \leq j < \text{size}(l_i)\}$ is the set of places (4.1)
- $E_{\mathcal{T}} \subseteq N \times N$ where $(T_{i_1,j_1}, T_{i_2,j_2}) \in E_{\mathcal{T}}$ exactly if $(l_{i_1}, l_{i_2}) \in E_{\mathcal{L}}$ and $j_2 = \left\lfloor \frac{j_1 \cdot \text{size}(l_{i_2})}{\text{size}(l_{i_1})} \right\rfloor$ (4.2)

The connection rule in definition (4.2) divides the size of two connected layers to calculate how many lower hierarchy layer places need to be connected to a higher hierarchy layer place. Each higher hierarchy layer place then is connected to that many lower hierarchy layer places. If a remainder results from the division, the initial places of the higher hierarchy layer are connected to an additional place of the lower hierarchy layer. Figures 4.5(d)–4.5(f) present the topology graphs that result with this definition for the layer trees of Figures 4.5(a)–4.5(c).

The transformation from layer trees to tool topology graphs creates tool layouts with properties that support scalable and easy to manage tools (without proof):

Lemma 1 (Structure) *Reverting the arc direction of $\mathcal{T}$ to a graph $\mathcal{T}' = (N, E'_{\mathcal{T}})$ with $(T_{i_1,j_1}, T_{i_2,j_2}) \in E'_{\mathcal{T}}$ exactly if $(T_{i_2,j_2}, T_{i_1,j_1}) \in E_{\mathcal{T}}$, yields a multitree, i.e., in $\mathcal{T}'$ for each node $T \in N$ holds that the set of reachable nodes of T forms an arborescence.*

Lemma 2 (Layer graphs and TBONs) *The topology graph $\mathcal{T}$ represents a TBON layout when $\mathcal{T}'$ as defined in Lemma 1 is an arborescence, which is exactly if the layer tree $\mathcal{L}$ has exactly one sink $l \in L$ with* $\text{size}(l) = 1$.

Lemma 3 (Outgoing communication) *For each node $T_{i,j} \in N$ holds* $\text{fanout}(T_{i,j}, \mathcal{T}) = \text{fanout}(l_i, \mathcal{L})$, *i.e., the number of outgoing arcs in the layer tree determines the number of outgoing primary communication direction channels of a node in the topology graph.*

Lemma 1 is crucial to make the layer tree to topology graph transformation practical for offloading-based tools. It guarantees that along the primary communication direction—the paths of the layer tree—exactly one path from an event source to a module instance exists, i.e., that the topology graph is a multitree [48]. Section 4.5 uses this property to derive an algorithm that preserves event order in the presence of aggregations.

Layer trees can construct TBON layouts (Lemma 2), which relates the proposed abstraction to infrastructures such as MRNet, SCI, and STCI. Intuitively, a layer tree creates a TBON layout if it is a list (has one sink) and if its last layer (the sink) has a layer size of 1. However, with the fixed connection rules (definition 4.2), there exist some TBON topologies that can't be created from layer trees. If necessary, additional connection rules can overcome this limitation. Also, some analyses can require global information, and may thus require mappings onto layers l with $\text{size}(l) = 1$. As a result, tool developers may need to restrict instantiations such that they meet such requirements. The GTI implementation includes functionality that can enforce or assure such restrictions, as to immediately identify invalid module mappings. At the same time, the tool topologies that result from the above definition do not have to use a root place. This supports topologies of some existing tools [84] that use an application layer and one/two additional hierarchy layers for tasks such as I/O forwarding. Pure TBON topologies would not support such use cases.

In addition, Lemma 3 aids for the instantiation system that handles the proposed abstraction, since it determines that each place has exactly as many outgoing arcs as it has child layers in the layer tree. With that, all places of each layer can execute similar event forwarding. This allows the instantiation system to determine event forwarding—that is based upon the subsequent event-flow definitions—once for each

layer and then to apply it to all places of this layer. Thus, the system does not need to generate specific event handling for each place.

The layer distribution rule (definition 4.2) connects places between two connected layers l_1 and l_2 ($(l_1, l_2) \in E_{\mathcal{L}}$) as equally as possible, but if $\text{size}(l_1) \bmod \text{size}(l_2) \neq 0$, then initial places of layer l_2 handle an additional place of layer l_1. Other distributions may provide a better balance or can adapt a layout such that it considers workload characterizations. GTI supports a second block-based distribution rule, where the function $\text{bsize} : L \to \mathbb{N} \setminus 0$ specifies the block size for each layer. For the block size distribution, for all layers l_1 and l_2 with $(l_1, l_2) \in E_{\mathcal{L}}$ must hold:

$$\big(\text{size}(l_2) - 1\big) \cdot \text{bsize}(l_2) < \text{size}(l_1) \leq \text{size}(l_2) \cdot \text{bsize}(l_2).$$

Block-based distribution replaces definition (4.2) with:

- $E_{\mathcal{T}} \subseteq N \times N$ where $(T_{i_1,j_1}, T_{i_2,j_2}) \in E_{\mathcal{T}}$ if $(l_{i_1}, l_{i_2}) \in E_{\mathcal{L}}$ and $j_2 = \left\lfloor \dfrac{j_1}{\text{bsize}(l_{i_2})} \right\rfloor$ (4.3)

The GTI implementation allows mixes of both distribution types for distinct pairs of connected layers, where the block distribution aids in application crash-handling (Section 4.3.5). The block-distribution supports process to compute core mappings that enable shared memory communication between some layers.

4.2.4 Communication System

The tool infrastructure abstraction defines which layers are connected and requires that events can be communicated between places of these layers. As a result, the layer tree suffices to define tool internal communication if an implementation of the abstraction provides some means of communication. At the same time, infrastructure approaches such as MRNet, which use a fixed means of communication have limited flexibility. Thus, the GTI implementation uses a flexible and exchangeable communication system instead. The following first describes this communication system. Subsequent definitions then formalize how the proposed abstraction considers this flexibility. The GTI implementation associates two types of communication modules with each pair of connected layers in the layer tree:

Protocol: A [*communication*] *protocol* module selects a communication medium, and

Strategy: A [*communication*] *strategy* module determines communication timing.

Protocol modules provide a flexible means of communication, where the implementation of GTI supports MPI-based communication [78], shared memory communication [126], and TCP socket communication.

Strategy modules decide how and when communication takes place. Strategy implementations in GTI include immediate, synchronous communication; immediate, asynchronous communication; and buffered, asynchronous communication. These different strategies allow tools to execute some of their analyses in the critical path (relevant for application crash handling), and to choose between latency or bandwidth efficiency. A study [78] details their implementations and compares their performance for a synthetic test case with MPI-based communication.

This pairing allows tool developers to flexibly adapt GTI-based tools. The protocol module selection may depend on the target computing system, and the strategy module selection may depend on properties of analysis modules or the need to handle a potential application failure. Figure 4.4 highlights these two module types. In the hierarchy of module instances that each place uses, protocol modules are children of strategy modules, such that the latter can utilize the former. Other modules that require communication services, e.g., modules for instrumentation, use services of strategy modules, rather than of protocol modules. With that, strategy modules have full control over communication timing, while their decisions remain transparent for other modules.

To formalize the use of these communication modules and their association to layer connections, let M_S be the set of strategy modules and M_P be the set of protocol modules. Given a layer tree $\mathcal{L} = (L, E_\mathcal{L}, l_0)$, the function $m_{\text{com}} : E_\mathcal{L} \to M_P \times M_S$ associates a pair of a protocol and a strategy module with each arc of the layer tree. Figure 4.9(a) (page 61) illustrates this association for a layer tree. It uses two types of protocols: *SM* to refer to a shared memory protocol and *MPI* to refer to a MPI-based protocol. The Figure omits the specification of a strategy module.

A further GTI specific formalization is the use of *place modules*, which manage tool places and form the root of their module instance hierarchies. These modules enable adaptions towards the mechanisms that create places in the GTI implementation. Their formalization uses a set of place modules M_D and a function $m_{\text{place}} : L \setminus \{l_0\} \to M_D$ that assigns a place module to each layer of tool places. The application layer uses no place module since the overall control flow on application places is with the application, rather than with the tool.

4.2.5 Hooks and Operations

An instrumentation system must observe the occurrence of relevant events. Hooks describe the sources for these events and the subsequent event-flow definitions define which hooks need to be instrumented for a specific tool layout. Additionally, the event-flow defines what information is required for events from each hook, what analyses need to be triggered for these events, and which tool internal event forwarding these events require. In the abstraction, hooks are functions of a specific arity, each of their arguments can form the input for an analysis; either directly or after preprocessing by an operation. Let H represents the set of hooks and let the previously defined arity function specify the number of arguments that each hook provides. The introductory example from Figure 4.3(a) uses $H = \{h_0, h_1\}$ with $\text{arity}(h_0) = 1$ and $\text{arity}(h_1) = 2$.

Operations allow tool developers to transform or preprocess arguments of hooks. Let O be the set of operations. Again let the arity function specify the number of arguments that each operation expects. Each operation returns a single argument that may be used as an input to an analysis. The introductory example from Figure 4.3(b) uses $O = \{o_0\}$ with $\text{arity}(o_0) = 2$.

The implementation of GTI represents analyses and hooks as functions of the programming language C. Operations in GTI are customizable templates (source snippets). As a result, GTI associates a programming language datatype with each argument of a function, analysis, or operation; such as to check whether mappings of analyses use compatible datatypes. In addition, hook arguments may be arrays or pointers. Thus, the GTI implementation supports array arguments as specifically flagged hook arguments. An additional argument, or the result of an operation, then provides the length of the array. Operations in GTI may also return arrays, in which case the operation both returns the resulting array and its length. For simplicity, this document handles all arguments as scalar values.

4.2.6 Mappings

Mappings interrelate the introduced terms of the abstraction. The analysis-hook mappings define which hooks provide input for what analyses and the layer-module mappings define which layers execute what modules.

4.2.6.1 Analysis-Hook Mapping

Analysis-hook mappings specify that an analysis is interested in events of a specific hook, i.e., they specify the inputs that particular tool analyses need. At the same time, this mapping must represent the structures that the introductory example in Figure 4.3(d) illustrates, to precisely define the mapping of hook arguments to analysis arguments. Additionally, this mapping must include any operations that provide input to an analysis along with its respective mapping. Sequences serve for this purpose; a mapping of an analysis to a hook is defined with the following steps:

- Seqs_{n_1,n_2} defines sets of valid mappings of an operation o with $\text{arity}(o) = n_1$ to a hook h with $\text{arity}(h) = n_2$;
- ASeqs_{n_1,n_2} defines sets of valid mappings of an analysis a with $\text{arity}(a) = n_1$ to a hook h with $\text{arity}(h) = n_2$;
- The set AH includes pairs of analyses and hooks to define which analyses are mapped to which hooks; and
- $m_{A,H}$ assigns an analysis mapping from ASeqs_{n_1,n_2} to each pair in AH, where the mapping must use n_1 and n_2 such that they are compatible with the analysis and hook pair.

The set of possible mappings of an operation $o \in O$ with $\text{arity}(o) = n_1$ to a hook $h \in H$ with $\text{arity}(h) = n_2$ is the set of sequences:

$$\text{Seqs}_{n_1,n_2} = \big\{(k_1, k_2, \ldots, k_{n_1}) : \forall i \in \{1, 2, \ldots, n_1\} \text{ holds } 0 \leq k_i < n_2\big\}.$$

As an example, consider the mapping of operation o_0 as part of the mapping of analysis $a_{1,0}$ to hook h_1 in Figure 4.3(d). The mapping of o_0 uses the sequence $(0, 1) \in \text{Seqs}_{2,2}$, which represents that the first argument of h_1 forms the first input of o_0 and that the second argument of h_1 forms the second input of o_0.

Based on the sets of operation mappings, the set of possible analysis mappings for an analysis $a \in A$ with $\text{arity}(a) = n_1$ to a hook $h \in H$ with $\text{arity}(h) = n_2$ is:

$$\begin{aligned}\text{ASeqs}_{n_1,n_2} = \big\{&(\text{in}_1, \text{in}_2, \ldots, \text{in}_{n_1}) : \forall i \in \{1, 2, \ldots, n_1\} \text{ holds} \\ &0 \leq \text{in}_i < n_2 \text{ or } \text{in}_i = (o \in O, \text{seq} \in \text{Seqs}_{\text{arity}(o),n_2})\big\}\end{aligned}$$

The set $\text{AH} \subseteq A \times H$ specifies the mappings of analyses to hooks as pairs of an analysis a and a hook h, which represents that analysis a is mapped to hook h. The function $m_{A,H} : \text{AH} \to \bigcup_{i,j \in \mathbb{N}} \text{ASeqs}_{i,j}$ specifies the argument mappings for each of these analysis-hook pairs. For a correct mapping, for all $(a, h) \in \text{AH}$ holds $m_{A,H}(a, h) \in \text{ASeqs}_{\text{arity}(a),\text{arity}(h)}$.

The analysis-hook mapping in the introductory example in Figure 4.3(d) uses:

- $\text{AH} = \big\{(a_{0,0}, h_o), (a_{1,0}, h_1), (a_{1,1}, h_1)\big\}$,
- $m_{A,C}(a_{0,0}, h_o) = (0)$,
- $m_{A,C}(a_{1,0}, h_1) = \Big(0, \big(o_0, (0, 1)\big)\Big)$, and
- $m_{A,C}(a_{1,1}, h_1) = (1)$.

The implementation of GTI provides two execution orders for each analysis-hook mapping. Since GTI considers function calls of a programming language as hooks, these two orders allow GTI-based tools to distinguish whether mapped analyses are executed for the arguments that are being passed into the function (*pre* order) or for the arguments that result from the function call (*post* order). In the case of MPI runtime verification, this allows a tool to differentiate between applying checks to input arguments of an MPI operation or retrieving results, e.g., on newly created MPI resources, from an MPI operation. The formalization here omits this mapping order for simplicity.

4.2.6.2 Module-Layer Mapping

A tool developer must be able to specify which layers of the tool hierarchy execute which parts of the tool functionality. A module-layer mapping supports this purpose and associates modules to layers. The function $m_{L,M} : L \to \mathcal{P}(M_T)$ then formalizes the mapping. Since the tool infrastructure will

map aggregation modules automatically, with a consideration of applicability, the mapping specifically restricts the freedom of the mapping to tool modules only, i.e., $\mathrm{dom}(m_{L,M}) = M_T$. The example in Figure 4.3(f) (page 40) uses $m_{L,M}(l_0) = \{m_0\}$, $m_{L,M}(l_1) = \emptyset$, and $m_{L,M}(l_2) = \{m_1\}$.

4.2.7 Event-Flow

The previous definitions formally introduced the terms that allow a tool developer to specify a tool layout, as well as the mappings that connect analyses with hooks and modules with layers. The following definitions reuse the symbols that the previous formalizations introduced, while the symbol list for this thesis on page 161 summarizes them. From the introduced terms, a tool instantiation system can generate a tool topology graph $\mathcal{T} = (N, E_\mathcal{T})$ with the rules in definitions (4.1) and (4.2). Algorithms in subsequent sections then introduce how such an instantiation system can compute which modules a place must execute. The module-layer mapping serves as the input for this processing and it must consider module dependencies, as well as aggregation modules. Let the function $m_{\mathcal{T},M} : N \to \mathcal{P}(M)$ specify this result that assigns a set of modules to each place[2]. The layout and the sets of modules to execute on each place define the structure of the overall tool, but not its workings. The event-flow rules in this section fill this gap and define the overall activity in the tool. Subsequent sections then extend these rules for event aggregation and an additional communication direction.

Based on a topology graph $\mathcal{T}$, sets of modules to execute on each place $m_{\mathcal{T},M}$, and the analysis-hook mappings $\mathrm{AH} \subseteq A \times H$, the event-flow defines:

- What events the instrumentation system of the tool must observe,
- Towards which other places a place must forward observed or received events, and
- Which analysis must be triggered when the instrumentation system observes an event or if a place receives an event from another place.

Thus, this section formally defines the event-flow of the proposed abstraction. The last of the above three actions is independent of the communication direction that is associated with a hook:

Event-Flow 1 (Trigger) *A place T that observes or receives an event for a hook h must trigger an analysis a of a module $m \in m_{\mathcal{T},M}(T)$ (with $a \in a_M(m)$) exactly if:*

- $(a, h) \in \mathrm{AH}$.

The workings of the instrumentation system as well as the event forwarding depend on the communication direction of a hook.

4.2.7.1 Primary Communication Direction

The primary communication direction forwards events from the application layer towards the highest hierarchy layer, i.e., towards sinks in the layer tree. Analyses of modules that are mapped onto a place should receive information—in the form of events—for all hooks to which they are mapped, whenever they are triggered by a predecessor place in the tool topology graph. Thus, informally, the instrumentation system on a place must observe a hook of the primary communication direction if: Itself or a successor along the primary communication direction executes a module with an analysis that is mapped to the hook. Similarly, a place that observes a hook or receives information on a hook, forwards an event if: A direct successor place—or a successor of that place—executes a module with an analysis mapped to the hook. Both of the above informal definitions use the notion:

Does a successor place require information on an event of a hook?

[2] Subsequent sections highlight that all places of a layer execute the same modules, but using the set of all places N as the domain of $m_{\mathcal{T},M}$ provides convenience in definitions.

This notion determines whether to observe or forward an event. The relation $\mathrm{requiresInformation} \subseteq N \times H$ formally defines it as $(T, h) \in \mathrm{requiresInformation}$ exactly if $\exists T' \in N, m \in M, a \in A$:

- $T' \in \{T\} \cup \mathrm{successors}(T, \mathcal{T})$,
- $m \in m_{\mathcal{T},M}(T')$,
- $a \in a_M(m)$,
- $\mathrm{dir}(h) = \mathrm{primary}$, and
- $(a, h) \in \mathrm{AH}$.

The relation includes pairs of a place and a hook if the place itself or a direct successor of the place has at least one mapped module with an analysis that is mapped to the respective hook. With that relation:

Event-Flow 2 (Observe) *A place $T \in N$ must observe (i.e., instrument) a hook $h \in H$ exactly if* $(T, h) \in \mathrm{requiresInformation}$.

Event-Flow 3 (Forward) *A place T must forward an event of hook h, which it observes or receives from another place, to a direct successor T' exactly if* $(T', h) \in \mathrm{requiresInformation}$ *($(T, T') \in E_{\mathcal{T}}$).*

These two definitions complete the basic event-flow for the primary communication direction—adding to the definition of triggering analyses (Event-Flow 1)—and formally define the workings of tool places.

4.2.7.2 Broadcast Communication Direction

As to support communication from higher hierarchy layers towards the application layer, the proposed abstraction includes the *broadcast* direction as a second communication direction. This direction uses a broadcast semantic, which is, if a place sends an event to a direct predecessor in the topology graph then it sends the same event to all other direct predecessors as well. A relation $\mathrm{requiresInformationRevert} \subseteq N \times H$ inverts the communication direction of the relation $\mathrm{requiresInformation}$ to define whether a place needs information on an event of a hook along the broadcast communication direction. This relation is defined as $(T, h) \in \mathrm{requiresInformationRevert}$ exactly if $\exists T' \in N, m \in M, a \in A$:

- $T' \in \{T\} \cup \mathrm{predecessors}(T, \mathcal{T})$,
- $m \in m_{\mathcal{T},M}(T')$,
- $a \in a_M(m)$,
- $\mathrm{dir}(h) = \mathrm{broadcast}$, and
- $(a, h) \in \mathrm{AH}$.

Based on this relation, the following two additional definitions complement the event-flow for the broadcast communication direction:

Event-Flow 4 (Observe-Broadcast) *A place $T \in N$ must observe (i.e., instrument) a hook $h \in H$ exactly if* $(T, h) \in \mathrm{requiresInformationRevert}$.

Event-Flow 5 (Forward-Broadcast) *A place T must forward an event of hook h, which it observes or receives from another place, to a direct predecessor T' exactly if* $(T', h) \in \mathrm{requiresInformationRevert}$ *(with $(T', T) \in E_{\mathcal{T}}$).*

In the introductory example from Figure 4.3 (page 40) with its illustrated layout in Figure 4.2, assume that hook h_0 serves for the broadcast direction (subsequent sections specify this with $\mathrm{dir}(h_0) = \mathrm{broadcast}$). As an example, if place $T_{2,0}$ observes h_0 then the *Forward-Broadcast* rule requires that this place forwards the resulting event to both $T_{1,0}$ and $T_{1,1}$, since both nodes have a predecessor place ($T_{0,0}$–$T_{0,3}$) in the tool topology graph that executes module m_0, which has an analysis ($a_{0,0}$) that is mapped to h_0. Thus, both $(T_{1,0}, h_0)$ and $(T_{1,1}, h_0)$ are in the relation $\mathrm{requiresInformationRevert}$.

4.2.7.3 Event Shape

The event-flow rules define which events each place observes and towards which other places it must forward events. The specific argument mappings that $m_{A,H}$ defines allow a tool to just include "necessary" information in an event. That is, if a hook provides an argument that no place requires, the tool can omit this argument from events. Subsequent sections detail this notion and formally define the algorithms that compute the shape of events in a tool instance.

4.2.8 Specifications

An implementation of the proposed abstraction—such as GTI—must provide a language to specify all of the previous notations. As in the case of GTI, which provides various extensions of the basic abstraction, such a language will also include additional notations in practice. In GTI, XML structures serve as the language to formulate all of the terms. Thus, a GTI tool developer provides his analyses, packaged as modules, and a set of XML files to describe an overall tool. The instantiation system of GTI then reads these XML files and applies the definitions and algorithms from this and the subsequent sections to create a tool instance. Document type definitions aid tool developers in the creation of these XML files, while GTI's instantiation system provides extensive error reports and warnings for malformed specifications.

Four types of specifications form the input of GTI's instantiation system: The *analysis specification* defines the set of analyses and their packaging into modules. A *hook specification* then defines hooks and the analysis-hook mappings. The *layout specification* defines the layer tree and the layer-module mappings. Finally, the *GTI specification* defines communication and place modules that implement parts of the infrastructure. The symbol list on page 161 categorizes the introduced notations of the abstraction into these four specification types to detail which specification includes which notations.

GTI's implementation supports multiple hook and analysis specifications for a single tool instantiation. In order to reuse existing tool components, tool developers can add existing hook and analysis specifications to their own ones. Also note that APIs such as MPI use a large number of function calls, which impacts the creation of hook specifications. Generators can create initial hook specifications based on an input, e.g., a header file. GTI's implementation provides such a generator that is tested with inputs for MPI and CUDA [119] API functions.

4.2.9 Instantiation

The proposed abstraction specifies the event-flow for a tool instance, but not how a specific tool infrastructure manages tool instantiation. This section provides an overview of how the GTI prototype reads and evaluates specifications in order to create a tool instance.

Per default, GTI triggers tool instantiation directly before it executes a target application. Especially if a tool provides scripts that automatically generate tool layouts for a specific application run, then an instantiation at execution time enables a high degree of usability. As an example, a GTI-based tool could provide a wrapper around the application/script that starts an MPI application to both incorporate the generation of a tool layout and to trigger tool instantiation. For front-end/back-end systems, the GTI instantiation system also supports instantiation prior to an application run.

GTI uses its instantiation system to create additional modules—called *wrapper* and *arrival* modules. These modules implement the use case specific behavior that the event-flow defines. Wrapper modules implement event-flow rules that specify that a place has to observe an event along with the trigger and forwarding rules that concern these events. The arrival modules then implement trigger and forwarding rules for events that a place receives. Figure 4.6 illustrates GTI's instantiation workflow. Boxes with folded edges represent files that include specification files in GTI's XML format and intermediate files of the workflow. Modules, as well as the target executable, use boxes with white background in the figure. Boxes with black background represent the three GTI generators *weaver*, *wrapper generator*, and *arrival generator*. Finally, arrows in the figure indicate input-output relationships. The box with gray background at the bottom summarizes the components that a tool instance uses. These components

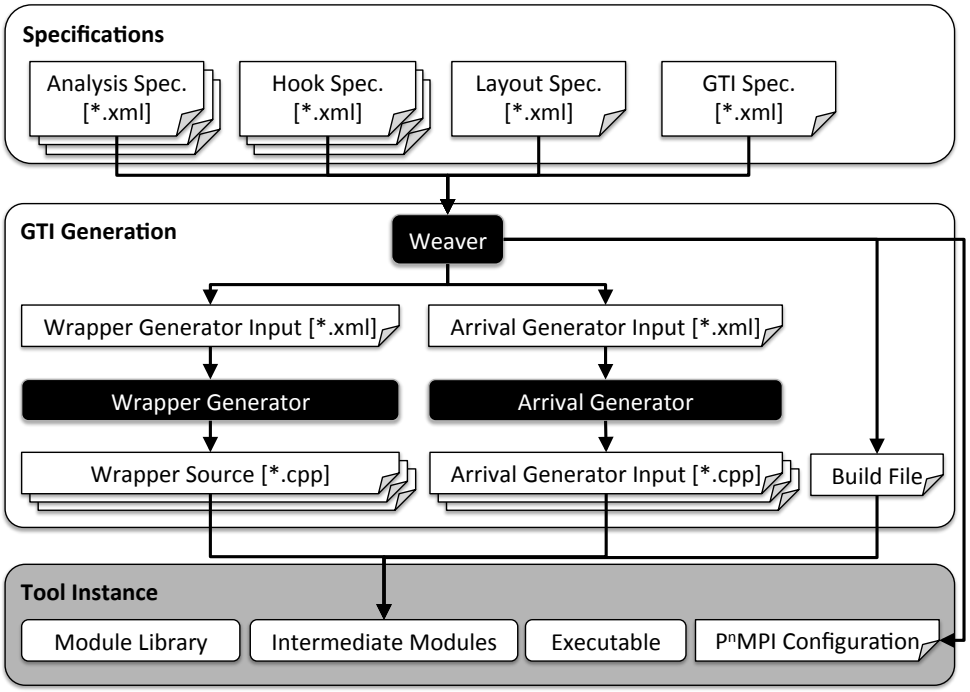


Figure 4.6: *The proposed abstraction requires a workflow to instantiate a tool from its specifications. The illustrated workflow highlights the handling within the GTI prototype.*

include GTI's module library, intermediate modules, the executable of the application, and a configuration file for P^{n}MPI. The module library includes modules for analyses, aggregations, communication protocols, communication strategies, and place modules. Thus, both the GTI implementation and the tool implementation provide these modules. The intermediate modules include the wrapper and arrival modules that implement the event-flow. Finally, a configuration file lists all modules in a format of the P^{n}MPI infrastructure. Additionally, this configuration file includes a description of the module hierarchy for the tool instance. Appendix A.1 (page 163) details the implicit module dependencies that non-analysis modules use along with an example module hierarchy.

GTI's instantiation process uses specification files as input. The *weaver* serves as the central generator that relates the specifications and applies the event-flow definitions. Additionally, the weaver computes module dependencies and automatically maps aggregation modules (subsequent sections). The primary outputs of the weaver are descriptions—in an XML format—for the wrapper and arrival modules. An additional build file then aids in the compilation and linking process for these intermediate modules. The *wrapper generator* and the *arrival generator* process the descriptions for the wrapper and arrival modules. These additional generators allow the weaver to remain independent of programming language, target programming paradigm, and instrumentation implementation.

In summary, GTI generates use case specific source code for wrapper and arrival modules, compiles these sources, and links the resulting objects into intermediate modules. A P^{n}MPI configuration file lists the modules that a tool instance uses and how they connect to form a module hierarchy.

4.3 Advanced Concepts

The previous section formally introduced the proposed abstraction and key choices for the GTI implementation. This section introduces an event injection concept that provides a basis for event aggregation. Additionally, this section introduces further types of communication directions. A scheme to handle an application crash then provides an important requirement for MPI runtime verification. Finally, this section presents an algorithm that serves as a driver for tool places.

4.3.1 Event Injection

The hook term of the proposed abstraction specifically avoids a restriction to just the application layer. Likewise, the event-flow rules consider events that result from hooks on all layers of the tool layout. Also, the illustrated instrumentation system of GTI uses a wrapper module on all layers. This serves for an important purpose:

The proposed abstraction allows hooks to be triggered on all layers.

Particularly, the application, as well as the tool itself, can trigger a hook. With that, hooks serve as a mechanism to allow a tool implementation to inject events.

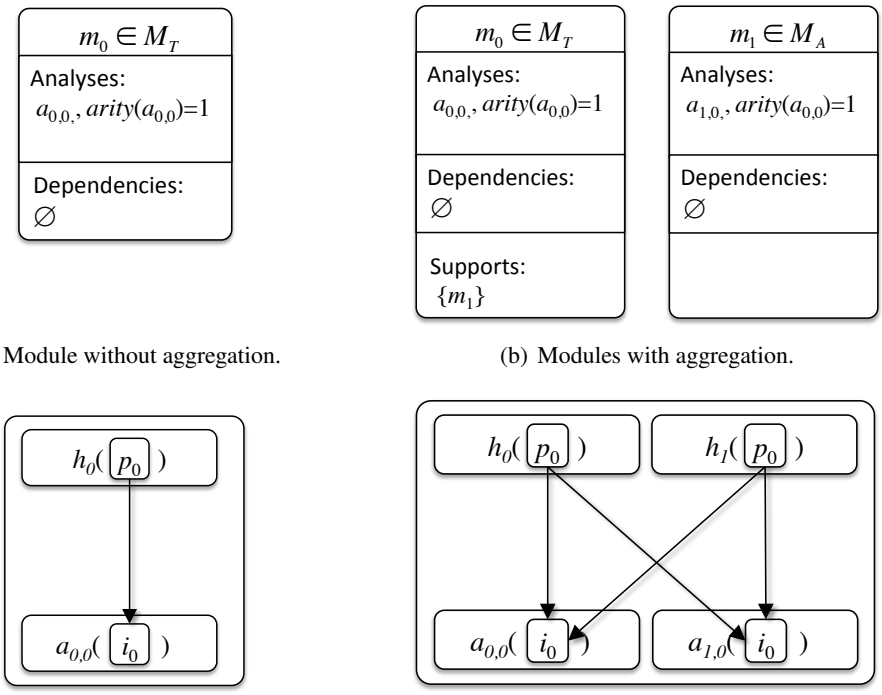
Event injection in the proposed abstraction supports use cases such as event filtering and event aggregation (subsequent sections). More generally, event injection allows any analysis to introduce a *response* event when it is triggered. Thus, computation in the proposed abstraction closely relates to active messages concepts [163, 167, 168]. Analyses can be considered as handlers of an active message system. The events—that hooks inject—then allow an analysis to trigger another analysis on a remote place. This notion reflects the active message concept of triggering handlers on remote processes. The key distinction here is that active message concepts usually enable a point-to-point style communication, whereas events in the proposed abstraction travel along communication directions. Especially, an event of the proposed abstraction can trigger multiple analyses, i.e., multiple handlers.

GTI incorporates event injection with a service mechanism that allows all analysis modules to gain access to the wrappers that its instantiation system generates. The implementation uses function pointers to provide this access. Thus, analyses can issue calls to such function pointers to inject a new event.

The proposed analysis-hook mappings enable an event-action mapping, i.e., they specify an action that handles a specific event. A more general version of this concept is *event-condition-action*, where an additional condition restricts the events to which the action applies. The analysis-hook mappings in the proposed abstraction do not incorporate such a condition, since the event injection concept provides a pattern to incorporate flexible conditions: If a hook h provides events and an analysis a is only interested in events from h that satisfy a specific condition. A direct analysis-hook mapping of a to h provides all events of h to the analysis, which includes events that do not satisfy the condition. If module placement requires event forwarding across layers for these events, then events that do not satisfy the condition introduce unnecessary overheads (assuming no other analysis is interested in these events). A second analysis $a_{\text{condition}}$ and a second hook $h_{\text{condition}}$ can avoid the forwarding of these superfluous events. The additional analysis implements the condition and is mapped to h. As to avoid event forwarding, the module that contains $a_{\text{condition}}$ should be mapped onto the layer that uses the hook h. If an event of h satisfies the condition, $a_{\text{condition}}$ injects a new event with the additional hook $h_{\text{condition}}$. Thus, $h_{\text{condition}}$ serves for injecting events that contain the information of the hook h and that satisfy the condition associated with a. The original analysis a is then mapped to $h_{\text{condition}}$ instead and will thus only perceive events that meet the given condition. This pattern provides a capability to apply conditions to the event-action mappings that the proposed abstraction provides, i.e., it provides an event-condition-action mapping.

4.3.2 Aggregations and Filters

Aggregation modules ($m \in M_A$) provide event aggregations in the proposed abstraction. Like any other module, aggregation modules use analysis-hook mappings to perceive their input events. In order to inject an aggregated event, these modules use event injection. Aggregation modules provide a spatial—as opposed to a temporal—aggregation. That is, aggregations combine events from multiple communication channels, as opposed to multiple events of a single channel. Aggregation modules decide at runtime which input events they use for an aggregation. Return values then provide the implementation of the tool infrastructure with feedback on aggregation decisions. This feedback allows the infrastructure to remove input events that were replaced with a new event at runtime, such as to ensure that no redundant events exist in the system.



(a) Module without aggregation.

(b) Modules with aggregation.

(c) Analysis-Hook mappings without aggregation.

(d) Analysis-Hook mappings with aggregation.

Figure 4.7: *An illustration of how an aggregation module can provide scalability for an analysis-hook mapping. Based on an existing analysis module m_0 in (a), along with its analysis-hook mapping in (c), an additional module m_1 in (b) can use a second hook (h_1 in (d)) to introduce an event aggregation.*

4.3.2.1 Illustration

As an example, consider the module m_0 from the introductory example in Figure 4.7(a). This module includes the analysis $a_{0,0}$ with an analysis-hook mapping to h_0, which Figure 4.7(c) illustrates. This hook could provide an integer number and m_0 could sum up *waves* of these numbers. Here, a *wave* refers to the set of events that results if each leaf place triggers the hook h_0 once, i.e., one event from each leaf place. The module with its analysis mapping will perceive all of these events if it is placed onto a layer with exactly one place. In that case it would become a scalability bottleneck.

As to remove this bottleneck, an aggregation can replace events from hook h_0, as to forward partial sums instead of the original values. Towards this end, a tool developer would add an additional aggregation module m_1, as well as an additional hook h_1, which serves for injecting aggregated events. Figure 4.7(b) presents the original module m_0 and the aggregation module m_1 with their analyses. In addition, Figure 4.7(d) illustrates the hook mappings of these two modules. To enable the aggregation, the analysis $a_{1,0}$ of module m_1 uses a mapping to hook h_0, since it uses this event as input for its aggregation. In addition, when m_1 creates an aggregated event with h_1, the tool forwards this event instead of the original events. Thus, the analyses of both m_0 and m_1 must be mapped to h_1, such that they perceive aggregated events. These analysis-hook mappings provide both the original module (m_0) and the aggregation module information on original and or aggregated events.

The previous example uses an additional hook h_1 for aggregated events. In practice, aggregations often modify the data that events carry to support the data transformation that applies to the original events. If both the original and aggregated events carry exactly the same information—as is likely in the previous example—tool developers can use a single hook for both the original and the aggregated events. In that case, the previous example would only use hook h_0, and the analyses of both modules would be mapped to this hook exclusively.

Event aggregations modify the event stream since they replace sets of events with new events—usually with a single event. Modules that are unaware of an aggregation may not function correctly as a result. Particularly, if they use no mapping to the hook that introduces aggregated events. To acknowledge that a module accepts the potentially alternative representation of an aggregated event, it must specify that it supports the aggregation module that injects this event. Function $s_A : M_T \to P(M_A)$ specifies which aggregation modules an analysis module supports. In the example of Figure 4.7(b), m_0 specifies that it supports the aggregation module m_1 with $s_A(m_0) = \{m_1\}$. Subsequent sections detail an algorithm that places aggregation modules automatically, such that it only maps them to layers that will provide input events for the aggregation, as well as to ensure that no aggregation removes input events for an analysis that does not support the aggregation.

4.3.2.2 Event-Flow

An aggregation module uses events of all hooks to which it is mapped as input, i.e., hooks to which any analysis of the module is mapped. The aggregation may replace any of these events, while it must not remove an input event of another module that does not support the aggregation module. Thus, an aggregation module may only be mapped onto a place if it does not share a mapping to a hook with another module of the same place, while the latter module does not support the aggregation. The relation $\mathrm{aggregationCompatible} \subseteq M_A \times \mathcal{P}(M)$ formalizes this notion and defines which aggregation modules are compatible with which sets of modules. It is defined as $(m_{\mathrm{agg}}, \mathrm{Modules}) \in \mathrm{aggregationCompatible}$ exactly if $\nexists m' \in \mathrm{Modules}, a, a' \in A, h \in H$:

- $m' \neq m_{\mathrm{agg}}$,
- If $m' \in M_T : m_{\mathrm{agg}} \notin s_A(m')$,
- $a \in a_M(m_{\mathrm{agg}})$,
- $a' \in a_M(m')$,
- $(a, h) \in \mathrm{AC}$, and
- $(a', h) \in \mathrm{AC}$.

The relation $\mathrm{aggregationCompatible}$ supports a formal definition of whether an aggregation module is compatible with a given set of modules. Following the definitions of the event-flow (Section 4.2.7 on page 49), for a topology graph $\mathcal{T} = (N, E_{\mathcal{T}})$ and a module mapping result $m_{\mathcal{T},M} : N \to \mathcal{P}(M)$, event aggregation requires the following adaptions and additions to the event-flow definitions:

Event-Flow 6 (Aggregation-Applicability) *For all places T, the set of modules that the place executes must be consistent with the relation* $\mathrm{aggregationCompatible}$*, i.e., for all modules $m_{\mathrm{agg}} \in m_{\mathcal{T},M}(T)$ with $m_{\mathrm{agg}} \in M_A$ must hold $\big(m_{\mathrm{agg}}, m_{\mathcal{T},M}(T)\big) \in \mathrm{aggregationCompatible}$.*

Event-Flow 7 (Aggregation-Replacement) *If an aggregation module $m_{\mathrm{agg}} \in M_A$ on a place T replaces a set of events $E = \{e_0, e_1, \ldots\}$ with injected events $E' = \{e'_0, e'_1, \ldots\}$, then:*

- *For each event in E', T triggers all applicable analyses; or alternatively, for each event in E;*
- *Analyses of aggregation modules form an exception, T always triggers them for the events in E; and*
- *For each successor place T' ($(T, T') \in E_{\mathcal{T}}$), T either forwards all events of set E or all events of set E' to T', the latter must only be used if:*
  - $\forall T'' \in \{T'\} \cup \mathrm{successors}(T', \mathcal{T}) : (m_{\mathrm{agg}}, m_{\mathcal{T},M}(T'')) \in \mathrm{aggregationCompatible}$

Event-Flow 8 (Aggregation-Trigger) *The original trigger rule (Event-Flow 1) is restricted as follows: A place T triggers analyses a of aggregation modules ($\exists m_{\text{agg}} \in m_{\mathcal{T},M}(T) : m_{\text{agg}} \in M_A, a \in a_M(m_{\text{agg}})$) only when it receives an event of a hook h with $(a,h) \in \text{AH}$ (but not when it observes h).*

The first rule (*Aggregation-Applicability*) defines when a place may execute an aggregation module, which is when the module is in the relation $\text{aggregationCompatible}$. The second rule (*Aggregation-Replacement*) then details how event aggregation must replace events in order to trigger analyses and forward events correctly. The important notion here is that when an aggregation module injects an aggregated event, then this event duplicates the information of its input events. If a tool would trigger analyses and forward events for both the input events and the aggregated events, then it would communicate redundant information and it would provide duplicated information to analyses, which could disturb their correct operation. Thus, a tool must apply event forwarding and analysis triggering either for the input events or for the aggregated event(s), but never for both. The *Aggregation-Replacement* rule exactly requires this behavior. However, a place may only forward the resulting events of an aggregation to a direct successor place if it and all of its successors are compatible with the aggregation. Finally, the third rule (*Aggregation-Trigger*) adapts the original triggering rule of the event-flow to only execute analyses of aggregation modules when a place receives events, but not when it observes events (i.e., not when the place itself triggers a hook). Since aggregation modules inject events to which their analyses are usually mapped themselves, e.g., see Figure 4.7(d), without this restriction, they would observe their own aggregation results.

4.3.2.3 Aggregation Module Feedback

The event-flow definitions for aggregation modules do not specify which events an aggregation replaces. Rather, aggregation modules can freely decide which events they replace with what events. This enables context sensitive aggregations that evaluate event information to decide which events qualify for aggregation. Also, for events with inconsistent information, as they can occur in the MPI runtime verification use case, aggregation modules can choose to not apply any aggregation to them. As a result, aggregation modules must provide feedback to the implementation of the tool infrastructure at runtime, in order to specify which events they replace. For the GTI implementation, place and arrival modules must consider this feedback and retrieve it from aggregation modules whenever an event triggers one of their analyses. Arrival modules use this information to decide whether they forward input events of an aggregation or the aggregated event(s) that they provide as a replacement. Also, if an aggregation applies to an event, arrival modules will try to trigger local analyses for the aggregated (more compact) event, rather than for the input events of the aggregation (freedom of the *Aggregation-Replacement* rule above). A subsequent section elaborates how place modules of GTI use feedback from aggregation modules to preserve event order in the presence of aggregations. Aggregation modules in the GTI implementation return the following information to provide the feedback that the arrival and place modules require:

$\text{closeChannel} \in \{\top, \bot\}$**:** The module started or continued an aggregation, but it requires additional events to complete;

$\text{finishedAggregation} \in \{\top, \bot\}$**:** The module completed an aggregation; and

channelListToOpen [set]**:** If an aggregation completes/fails it provides a list of so called channel identifiers to identify the events that it replaced or tried to replace.

As an example, consider the introductory example in Figure 4.7(a) with its binary TBON layout in Figure 4.2. The previously introduced aggregation module m_1 would execute on layer l_1, in order to aggregate events from hook h_0 on places $T_{1,0}$ and $T_{1,1}$. Note that subsequent sections introduce an algorithm that places aggregation modules without violating any of the event-flow definitions. If the aggregation module m_1 replaces waves of events from h_0, then a possible aggregation on $T_{1,0}$ is:

- The analysis $a_{1,0}$ of the module instance of m_1 is triggered for an event of h_0 that originates from $T_{0,0}$:
  - m_1 starts an aggregation, but can't finish it since it waits for an event from $T_{0,1}$ to complete the portion of the wave that it can observe, thus, it returns: $\mathrm{closeChannel} = \top$, $\mathrm{finishedAggregation} = \bot$, and $\mathrm{channelListToOpen} = \emptyset$:
- The analysis $a_{1,0}$ of the module instance of m_1 is triggered for a second event of h_0 that originates from $T_{0,1}$:
  - m_1 continues and finishes its aggregation by injecting an event with hook h_1, thus, it returns: $\mathrm{closeChannel} = \bot$, $\mathrm{finishedAggregation} = \top$, and $\mathrm{channelListToOpen} = \{c\}$:

This example uses the identifier c to identify the channel of the first event, later sections introduce the structure of these identifiers. The return values in the above example illustrate status information on the aggregation, such that the tool infrastructure can ensure that it correctly follows the freedom of the *Aggregation-Replacement* rule. That is, for a successful aggregation, arrival modules discard the input events of the aggregation—the ones that $\mathrm{channelListToOpen}$ identifies—and applies analyses and event forwarding to the aggregated event(s) instead.

Additionally, the aggregation module feedback allows aggregations to fail. In the use case of MPI runtime verification, if aggregations apply to incorrect MPI operations, then these aggregations should be able to abort a started aggregation once an inconsistency occurs. For that purpose, if an aggregation fails to successfully complete an aggregation, it lists the events that it tried to aggregate in $\mathrm{channelListToOpen}$ and returns $\mathrm{finishedAggregation} = \bot$. An arrival module then applies analyses and event forwarding to the events that $\mathrm{channelListToOpen}$ identifies.

Aggregation modules that do not inject events when they return $\mathrm{finishedAggregation} = \top$ implement an event filter. Thus, aggregation modules both serve for event aggregation and hierarchical filters.

GTI's implementation also supports a timeout mechanism to abort aggregations if a waited-for event does not arrive within a predefined timeframe. This functionality also supports the MPI runtime verification use case in which events that should occur, e.g., all processes in a communicator issue the same collective operation, may not occur due to a defect in a program. If a timeout occurs, GTI notifies all aggregation modules that their current aggregation failed. Afterwards, arrival modules apply analyses and event forwarding to the input events that any aborted aggregation used. Place modules realize this timeout mechanism in the GTI implementation.

4.3.3 Broadcasts

The arcs of the layer tree specify the primary communication direction in the proposed abstraction. However, some tools require additional communication directions. As an example, to notify lower level hierarchy layers of the results of an analysis, a tool would communicate in the opposite direction of the layer tree arcs. The *broadcast* communication direction serves for this purpose and the event-flow definition of the previous sections introduced its event forwarding and analysis trigger rules already.

Leaf places—application processes—cannot use the broadcast communication direction. Rather, tool places utilize this communication direction. They use event injection to trigger hooks that are associated with the broadcast direction. The proposed abstraction assigns each hook a communication direction with $\mathrm{dir} : C \rightarrow \{\mathrm{primary}, \mathrm{broadcast}, \mathrm{intralayer}\}$. $\mathrm{Primary}$ serves for communication along arcs of the layer tree, $\mathrm{broadcast}$ for the broadcast direction, and $\mathrm{intralayer}$ for communication within a layer (subsequent section).

Leaf places do not use a driver module as to return the control flow to the target application. Thus, the GTI implementation only broadcasts events towards tool places. If GTI would transfer events to leaf places, it would need to ensure that these places receive these events. The use of an extra thread on each application process would simplify such designs. Within GTI's abstraction, such an extra thread could be integrated as a tool place. Also, additional threads may limit portability, e.g., as experience within the MRNet project indicates [88].

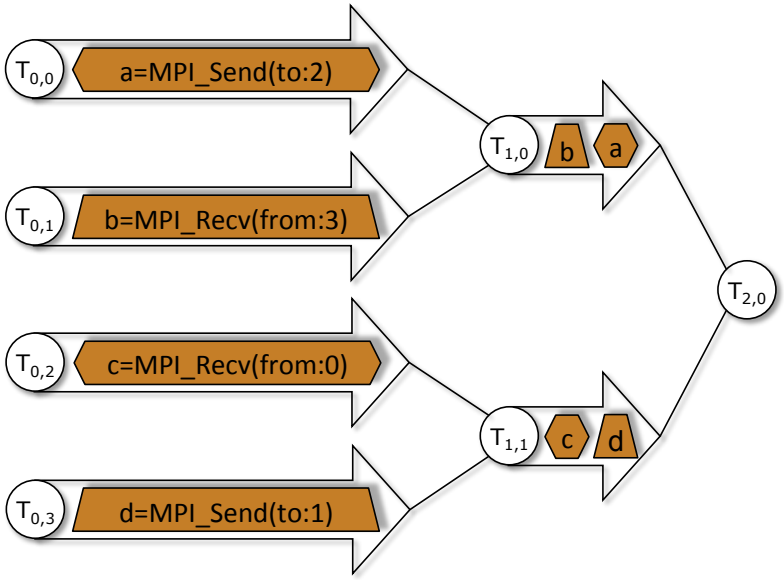


Figure 4.8: *For a filter-based implementation of MPI point-to-point matching, a situation as in this illustration can cause load imbalance.*

4.3.4 Intralayer Communication

Runtime verification tools for MPI must match point-to-point messages for analyses such as datatype matching, e.g., to detect the type matching defect in the example of Figure 2.1 on page 8. Matching such pairs of events (send and receive) can create load imbalances in a purely hierarchical tool, which could limit the scalability of such an approach.

4.3.4.1 Motivation

The matching of a pair—send and receive—of point-to-point communication operations requires that a place receives information on both operations. As an example, a TBON-based message matching tool [72] uses an event filter to implement point-to-point matching. For this approach, each place matches and filters out all point-to-point events for which it can receive both the send and the receive operation. Figure 4.8 illustrates this concept with a tool layout of three layers, four leaf places, two places in the intermediate layer, and one root place. Place $T_{1,0}$ would match and filter out all events of point-to-point operations that transfer data between places $T_{0,0}$ and $T_{0,1}$. The root place $T_{2,0}$, can observe events from $T_{0,0}$–$T_{0,3}$ and would thus match/filter all point-to-point events that transfer data between these four places, and which have not been matched by another place on a lower-level hierarchy layer.

The figure illustrates a mix of point-to-point operations for which the places on the intermediate tool layer cannot filter/match any events. The event shape—hexagon shape and trapezium shape—highlights the matching point-to-point operations in the figure. Both $T_{1,0}$ and $T_{1,1}$ cannot observe both events of either pair. Thus, the root $T_{2,0}$ must receive and match all events for this example. At scale, such imbalances can increase tool overheads or even limit scalability.

In general, for p application processes, the binomial coefficient $\binom{p}{2}$ represents the amount of process pairs that can exchange point-to-point messages with each other (excluding communication of application places with themselves). Without a-priori information on the communication pattern of the target application, tools should try to distribute load equally across tool places. This follows the assumption that the likeliness of all communication pairs is equal. In that case, each tool place should cover an equal share of the total amount of process pairs.

However, consider tool layouts that are k-ary trees, i.e., trees where all non-leaf places have fan-in k. A first layer place covers $\binom{k}{2}$ leaf place pairs, whereas a second layer place covers $\binom{k^2}{2}$ leaf places of which

its ancestor places can handle $k \cdot \binom{k}{2}$ pairs. For k-ary trees, the function $\text{firstPairs} : \mathbb{N}\setminus\{0\} \times \mathbb{N}\setminus\{0\} \to \mathbb{N}$ generalizes this property for a place on layer $i > 0$:

$$\text{firstPairs}(i,k) = \binom{k^i}{2} - k \cdot \binom{k^{i-1}}{2}$$

To satisfy an about equal share of covered pairs, a higher-level place should cover about as many pairs as a lower-level place, but:

$$\text{balance}(k) = \frac{\text{firstPairs}(i,k)}{\text{firstPairs}(i-1,k)} = \frac{\binom{k^i}{2} - k \cdot \binom{k^{i-1}}{2}}{\binom{k^{i-1}}{2} - k \cdot \binom{k^{i-2}}{2}} = k^2$$

Thus, for k-ary tree layouts, a higher-level place covers k^2 as many pairs as its child places. As an example, consider the binary tree layout in Figure 4.8. In this layout, places on the first tool layer cover 1 pair, whereas the root place already covers 4 distinct pairs that no child place covers. Thus, a filtering-based point-to-point matching approach in a TBON layout strongly diverges from the goal of distributing process pairs equally. Thus, it may cause load imbalance for some communication patterns.

The layer trees of the proposed abstraction allow a duplication of higher hierarchy layers as to mitigate the effect of $\text{balance}(k)$. However, to derive equal amounts of covered pairs, each place in a hierarchy layer must be duplicated k^2 times in the next higher hierarchy layer. Thus, degenerating the hierarchy. At the same time, layouts that avoid additional hierarchy layers—e.g., with layer trees where all tool layers are directly connected to the application layer—provide no hierarchy for event aggregation.

4.3.4.2 Intralayer Direction

Following the previous discussion, an exclusive use of the primary communication direction is unsatisfactory for point-to-point matching of applications with arbitrary communication patterns. Thus, the proposed abstraction provides the intralayer communication direction that implements communication between places of the same layer. This communication uses point-to-point semantics, such that a place can communicate with any place in the same layer. Assume that event pairs of leaf places have an a-priori distinction, e.g., for point-to-point communication one event is a send and the other is a receive. Tool places with intralayer communication receive both event types for all their connected leaf places. While they analyze all events of the one type (e.g., receive), they forward events of the other type (e.g., send) to the place (same layer) that receives the matching event (e.g., the matching receive). In the example of Figure 4.8, $T_{1,0}$ could forward the send event a to $T_{1,1}$ and $T_{1,1}$ could forward the send event d to $T_{1,0}$. With that, both $T_{1,0}$ and $T_{1,1}$ can match one point-to-point pair each. As a result, intralayer communication allows modules on tool places to replay the point-to-point communication pattern of the target application as part of their analysis. Existing approaches also use point-to-point communication for tasks such as message matching, where VampirServer [21] uses MPI-based communication during trace visualization and Scalasca [53] uses MPI-based communication to detect inefficiency patterns for performance analysis.

Intralayer communication is not necessary for all layers of a layout; rather the tool developer specifies which layers should use this communication system. For a layer tree $\mathcal{L} = (L, E_\mathcal{L})$ with root l_0, the layout specification provides a set $L_{\text{intra}} \subseteq L \setminus l_0$ that specifies the layers that use intralayer communication. Each layer in L_{intra} uses a module pair to select a communication strategy and a communication protocol. The function $m_{\text{icom}} : L_{\text{intra}} \to M_P \times M_S$ specifies this selection. The GTI implementation lets protocol modules specify whether they support intralayer communication, and uses specific intralayer communication strategies that can handle increased numbers of communication channels [75].

4.3.4.3 Intralayer Event-Flow

Following the definitions of the event-flow (Section 4.2.7 on page 49), for a topology graph $\mathcal{T} = (N, E_{\mathcal{T}})$ and a module mapping result $m_{\mathcal{T},M} : N \to \mathcal{P}(M)$, the intralayer communication direction requires the following additional event-flow definitions for hooks $h \in H$ with $\mathrm{dir}(h) = \mathrm{intralayer}$:

Event-Flow 9 (Intralayer-Observe) *A place $T \in N$ of a layer $l \in L_{\mathrm{intra}}$ must observe h exactly if $\exists m \in M, a \in A$:*

- $m \in m_{\mathcal{T},M}(T)$,
- $a \in a_M(m)$,
- $(a, h) \in \mathrm{AH}$.

Event-Flow 10 (Intralayer-Trigger) *As an exception to the analysis trigger rule that applies to the primary and broadcast communication directions (Event-Flow 1), a place triggers analyses a ($\exists m \in m_{\mathcal{T},M}(T) : a \in a_M(m)$) with $(a, h) \in \mathrm{AH}$ only when it receives an event for hook h (but not when it observes it).*

Event-Flow 11 (Intralayer-Forward) *When a place T observes h, it forwards an event to a place T' of layer l; The hook h must identify the target place T'.*

The above definition for observing events of intralayer hooks does not consider successor/predecessor places in the topology graph, as for the primary or broadcast communication directions. Rather, the definition checks whether a module on the place itself has a mapping to this hook. Since all places of a layer execute the same modules, this imposes that other places on the same layer execute a module with an analysis mapping to the intralayer hook. In that case, a place must observe the hook.

Both the primary and the broadcast communication direction trigger analyses both when they receive and when they observe an event. Executing analyses when a place observes an intralayer hook would only provide redundant information to the observing place, since it triggered the hook itself. Thus, the definition above restricts the triggering of analyses to event receival.

Finally, the forwarding rule for intralayer hooks specifies that the hook must provide information on which place should receive the event. In the GTI implementation, the last argument of an intralayer hook h provides this target place identifier. Accessory functions of GTI's runtime provide information on place identifiers, e.g., to determine which place receives events for a specific leaf place.

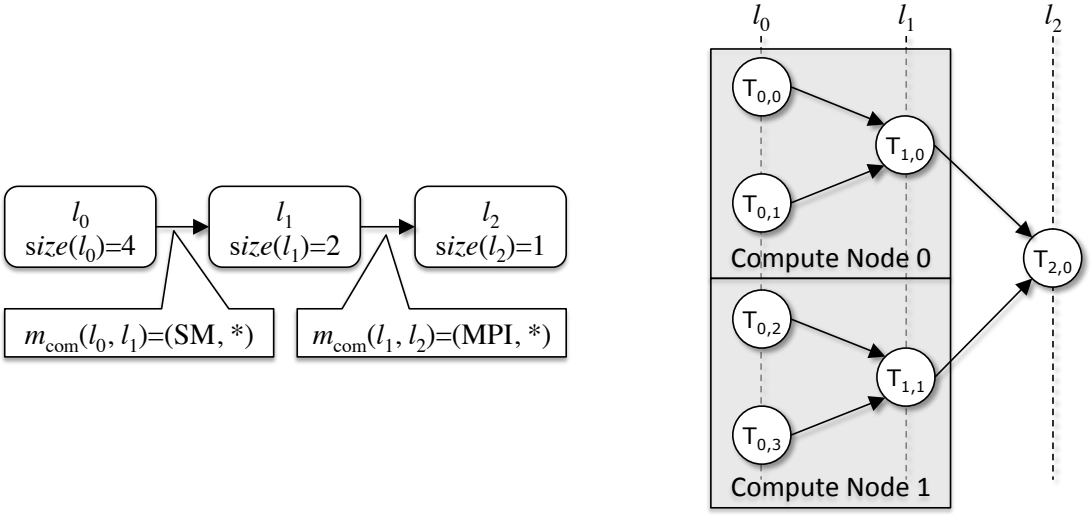
4.3.4.4 Alternatives

Intralayer communication emulates the flexibility of a direct point-to-point communication layer. If an approach with this increased connectivity is undesirable/impractical for a target programing paradigm or a target compute system, alternatives such as tree reconfigurations can adapt a TBON layout such that sender and receiver processes connect to the same first layer tool place. Approaches such as the resilient TBON services in MRNet [6], and adaptive reorganizations of overlay networks, e.g., XPORT [121] and OMNI [13], could provide such reconfigurations. However, the frequency with which an MPI application changes its communication pattern influences the applicability of these approaches.

4.3.5 Crash-Handling

Support to handle an application crash is a requirement for the runtime verification use case. GTI provides crash-handling as part of its implementation and requires no modifications to the proposed abstraction. In case of an application crash, a crash-handling scheme must guarantee to [126]:

- Avoid the loss of event information,
- Ensure that all tool analyses can continue, and
- Existing event information can still be communicated within the tool.



(a) A layer tree with protocol selections highlights the use of shared memory communication as an alternate means of communication.

(b) An illustration of a tool layout and its mapping onto compute nodes highlights the availability of shared memory between places of the application layer and the first tool layer (for the layer tree in (a)).

Figure 4.9: *The use of an alternate means of communication enables an application crash-handling scheme in the GTI prototype.*

A study [126] summarizes potential approaches to satisfy these conditions, as well as the approach that the GTI implementation uses. The following shortly summarizes and illustrates this approach. GTI employs an alternate means of communication to enable tool internal communication even if an application crash occurred. Signal and error handlers then ensure that the execution of all places continues after a crash.

GTI tool layouts that tolerate an application crash use a shared memory communication protocol between the leaf places and their connected tool places. All other layer connections use an MPI-based communication protocol. Figure 4.9(a) presents a layer tree with the communication protocols of GTI's crash-handling scheme. The figure uses SM to refer a shared memory protocol and MPI to refer to an MPI protocol. When GTI instantiates a tool layout from a layer tree, it must map places to compute cores. The mapping must ensure that connected application and tool places can use shared memory communication, i.e., execute on the same compute node. The block-based layer distribution (Section 4.2.3 on page 44) supports such mappings for resource allocation systems that place processes in a node-core order. The node-core order fills compute cores of a compute node first and continues with the next node only after all cores of a node are used up. System specific resource allocation switches can enforce a placement in node-core order if necessary. Figure 4.9(b) illustrates such a mapping of a tool layout onto compute nodes/cores. It uses the layer tree in Figure 4.9(a) as input and ensures that leaf places and their connected tool places execute on the same compute nodes. The example uses compute nodes with three cores each.

If all communication strategies send their events immediately, i.e., apply no buffering, then the use of an alternative means of communication suffices to process all events that occur prior to an application crash. Additional tool internal communication [126] supports crash handling with aggregating communication strategies in GTI.

4.3.6 Place Modules and Shutdown

In the GTI implementation, place modules manage the control flow of tool places. While the event-flow definitions of the abstraction define the activities of tool places, they do not define how places realize them. Figure 4.10 provides pseudo code for a simplified main loop of a place module. The

```
Function placeDriver
1  aliveCount := toLeavesStrategy.getFanIn() + successorSinkCount()
2  while aliveCount > 0 do
       /* Choose a communication direction to handle                    */
3      direction := choose({primary, broadcast, intra})
4      switch direction do
5          case primary
               /* Try to unqueue an event, else try to receive one      */
6              (hasEvent, e, cId) := channelTreeQueues.findEventToDequeue(ϵ)
7              if hasEvent = ⊥ then
8                  if toLeavesStrategy.isMessageAvailable() then
9                      e := toLeavesStrategy.receiveMessage()
10                     cId := arrival.getEventChannelId(e)
11                     if channelTreeQueues.canProcess(cId) then
12                         hasEvent := ⊤
13                     else
14                         channelTreeQueues.enqueue(e, cId)
               /* If event available: Process, update channelTreeQueues,
                  and check whether it is an initialize shutdown event  */
15             if hasEvent then
16                 (closeChannel, channelListToOpen) :=
                   arrival.processPrimaryDirectionEvent(e, cId)
17                 if closeChannel = ⊤ then
18                     channelTreeQueues.suspendChannel(cId)
19                 for openId ∈ channelListToOpen do
20                     channelTreeQueues.openChannel(openId)
21                 if arrival.isInitializeShutdown(e) then
22                     aliveCount := aliveCount − 1
23             break
24         case intra
25             if intraStrategy.isMessageAvailable() then
26                 e := intraStrategy.receiveMessage()
27                 arrival.processIntraEvent(e)
28             break
29         case broadcast
30             strategy := choose(toRootStrategies)
31             if strategy.isMessageAvailable() then
32                 e := strategy.receiveMessage()
33                 arrival.processBroadcastEvent(e)
                   /* Check whether it is a shutdown notification       */
34                 if arrival.isNotifyShutdownEvent(e) then
35                     aliveCount := aliveCount − 1
36             break
```

Figure 4.10: *The main loop of place module must receive events from all communication directions, consider the state of ongoing event aggregations in order to preserve event order (Section 4.5), and observe control messages to apply a coordinated tool shutdown.*

pseudo code excludes the timeout handling that aborts aggregations and focuses on how a place module interacts with arrival modules, and communication strategy modules. This includes one strategy—the toLeavesStrategy—that provides events that progress along the primary communication direction to the place. Further, the intraStrategy provides events that use the intralayer communication direction. For each successor layer in the layer tree, one strategy—a toRootStrategy—provides events that progress along the broadcast direction to the place. The handling of new events highlights how a place uses data structures—the channelTreeQueues—for order preserving event aggregation. Subsequent sections detail these data structures and the algorithms that apply to them. Finally, a place module must provide a shutdown mechanism that is triggered when all the leaf places that are connected to the place indicated their termination. This section introduces a two-phase algorithm that provides a default shutdown handling, but that also provides a mechanism to implement tool specific shutdown conditions. In summary, the following variables represent data structures or dependent modules of a place module:

- arrival: The arrival module that triggers analyses and forwards newly received events,
- toLeavesStrategy: The communication strategy that connects towards lower hierarchy layers,
- toRootStrategies: Communication strategies that connect towards higher hierarchy layers,
- intraStrategy: The communication strategy for the intralayer direction,
- channelTreeQueues: A queuing and aggregation state data structure for order preserving event aggregation (subsequent sections), and
- aliveCount: Determines when a tool place can exit its main loop.

The algorithm in Figure 4.10 chooses a communication direction per iteration to handle one of the three available directions (line 3). The function choose reflects this decision and a round-robin technique can implement it. Driver implementations could also use multiple threads to handle communication directions in parallel. Event processing for the primary direction requires a queuing technique that applies the channelTreeQueues data structure to implement order preserving event aggregation. Subsequent sections detail this data structure and the queuing rules that it applies, while this section discusses the respective handling that place drivers use to adapt to these decisions. Before the algorithm receives a new event it checks whether any event was previously queued and can now be processed (line 6). Otherwise, the algorithm checks whether an incoming message provides a new event for the primary direction (line 8). If so, it receives the new event and retrieves a channel identifier that influences queuing decisions for order preserving event aggregation (lines 9–11). If a place can process a newly received event or could dequeue a previously queued event, it passes this event to the arrival module and updates the state of channelTreeQueues (lines 15–23). The other two communication directions require no event queuing and directly pass newly received events to the arrival module (lines 25–28 and 30–36).

Application places shut down when they observe a specifically marked shutdown hook $h \in H$. The main loop of the algorithm in Figure 4.10 runs until the aliveCount is 0. A place T of a topology graph $\mathcal{T} = (N, E_{\mathcal{T}})$ initializes this count with $\text{fanin}(T, \mathcal{T}) + \text{successorSinkCount}(T, \mathcal{T})$. The first term calculates the number of places that forward information along the primary communication direction to T. The second term—successorSinkCount$(T, \mathcal{T})$—then calculates the number of descendant places that are a sink in the topology graph, it is defined as:

$$|\{T' \in N : T' \in \text{successors}(T, \mathcal{T}) \text{ and } T' \text{ is a sink in } \mathcal{T}\}|$$

The aliveCount serves for a two phase shutdown handling protocol:

In the first phase places forward events of the shutdown hook h along the primary communication direction. An implicit aggregation replaces all incoming events for h with a single outgoing event on each place. Thus, each tool place T receives exactly $\text{fanin}(T, \mathcal{T})$ events of type h. Lines 20–21 decrement the aliveCount when a place receives an event of type h. When the last of these events arrives, a place T has an aliveCount of successorSinkCount$(T, \mathcal{T})$.

In the second phase places that are sinks in the topology graph and that received their last event of type h broadcasts a new event h'. An implicit GTI module triggers h'. Places of sink layers then exit their main loop, since their $\mathrm{aliveCount}$ reached 0. Tool places that receive h' decrement their $\mathrm{aliveCount}$ (lines 33–34) and forward h' along the *broadcast* direction.

Thus, tool places exit their main loop after all their descendant places finished the first shutdown phase and forwarded an event of type h' (one for each sink among the descendants). In the case of a TBON layout, exactly one such sink exists. That is, the root of the layout injects and broadcasts an event of type h' to all tool places to notify them that all application places issued their shutdown hook h. This two-phase handling supports tools that inject additional events, such as that they can apply tool specific shutdown conditions. Event injection in the proposed abstraction allows an analysis to trigger further analyses, which can trigger additional analyses in turn. Thus, places can trigger additional analyses even if the application already issued a shutdown hook. This relates to active message frameworks that also require specific shutdown handling, e.g., Active Peebles [168] allows developers to specify termination conditions. GTI follows this approach and also provides tool developers an option to implement a tool specific shutdown trigger.

GTI provides two options to flag an application hook h as a shutdown trigger. In both cases the event shuts down application places. The first option lets tool places handle events of h as described above, i.e., events of h initiate the tool shut down. The second option lets tool places not apply shutdown handling to events of type h. This allows GTI tools to use an additional—tool specific—shutdown event that the tool injects in all of its first layer tool nodes. GTI then applies the shutdown handling above to this tool specific event. This technique allows tools to specify their own shutdown conditions and to impose synchronization that ensures that a tool analyzes all events prior to shut down.

4.4 Weaver Algorithms

Based on the event-flow of the proposed abstraction, this section details key algorithms in the weaver component of GTI's instantiation system. The event-flow definitions specify the activities of each place, but they do not specify the shape of events or how to compute which aggregation modules to execute on which layers. Thus, this section details algorithms for these purposes.

4.4.1 Module and Aggregation Placement

Figure 4.11 summarizes the module placement algorithm of GTI's weaver component. The weaver issues the algorithm with the root layer as input argument. The algorithm then uses a depth-first recursion (line 3) and applies a greedy scheme to add aggregation modules whenever potentially possible. For each layer l, the algorithm uses a set $\mathrm{Modules}(l) \subseteq M$ as its result set of modules to execute on layer l. Particularly, the algorithm uses the same set of modules on all places of a layer, which simplifies event routing and shaping. The module sets for each layer then provide the module-to-place association $m_{\mathcal{T},M} : N \to \mathcal{P}(M)$ that the previous sections used to define the event-flow (if $T \in N$ belongs to layer l then $m_{\mathcal{T},M}(T) \stackrel{def}{=} \mathrm{Modules}(l)$).

For its current layer l, the algorithm in Figure 4.11 first initializes the module set for this layer ($\mathrm{Modules}(l)$) as the set of modules mapped onto the layer (line 1) and then uses three steps:

Step 1: Add all aggregation modules supported by a successor layer to $\mathrm{Modules}(l)$, if they are compatible with all successor layer modules;

Step 2: Add all dependent modules and supported aggregations for modules in $\mathrm{Modules}(l)$; then

Step 3: Remove any aggregation module in $\mathrm{Modules}(l)$ that is incompatible with an analysis module in this set.

Function `placeModules(`l`)`

```
1  Modules(l) := m_{L,M}(l)
   /* Step 1: Consider aggregation modules of successor layers     */
2  for l' with (l, l') ∈ E_L do
3      placeModules(l')
       /* Gather all successor analysis and aggregation modules    */
4      DescModules := ∅
5      for l'' ∈ {l'} ∪ successors(l', T) do
6          DescModules := DescModules ∪ Modules(l'')
7      DescAggregations := DescModules ∩ M_A
       /* For each successor aggregation module, check whether it is
          compatible with all successor analysis modules           */
8      AggregationForwards(l, l') := ∅
9      for m_agg ∈ DescAggregations do
10         if (m_agg, DescModules) ∈ aggregationCompatible then
11             Modules(l) := Modules(l) ∪ {m_agg}
12             AggregationForwards(l, l') := AggregationForwards(l, l') ∪ {m_agg}
   /* Step 2: Add supported aggregations and module dependencies   */
13 while ∃m ∈ Modules(l), m' ∉ Modules(l) : m' ∈ d_M(m) ∨ m' ∈ s_A(m) do
14     Modules(l) := Modules(l) ∪ {m'}
   /* Step 3: Remove incompatible aggregations (greedy scheme)     */
15 while ∃m_agg ∈ Modules(l) : m_agg ∈ M_T ∧ (m_agg, Modules(l)) ∉ aggregationCompatible do
16     Modules(l) := Modules(l) \ {m_agg}
17     for l' with (l, l') ∈ E_L do
18         AggregationForwards(l, l') := AggregationForwards(l, l') \ {m_agg}
```

Figure 4.11: *An algorithm to place modules onto layers and to determine which aggregation modules are applicable for which layer tree connections.*

The first step of the algorithm iterates over all direct successors l' of the current layer to determine which aggregation modules could support the modules of l' or any of its successors. This choice employs the freedom of the *Aggregation-Replacement* rule (Event-Flow 7) for aggregations. which is that a place can forward aggregated events to some direct successors, while it may forward original events to others. Instead of only forwarding aggregated events—and employing the respective aggregation module for this aggregation—only if all successors of the current layer support this aggregation module, a layer-based choice increases the number of aggregations that a layer can execute. The algorithm uses the sets $\mathrm{AggregationForwards}$ to store which aggregation modules may forward aggregated events to which direct successors of a layer. For each direct successor l' of the current layer, lines 4–7 determine the set of all modules that execute on l' and its successor layers as $\mathrm{DescModules}$. An additional set $\mathrm{DescAggregations}$ then limits the set of all descendant modules to just aggregation modules. For each of these aggregation modules, lines 9–12 check whether the aggregation module is applicable for the set $\mathrm{DescModules}$ with a comparison to the relation $\mathrm{aggregationCompatible}$, which was introduced for the event-flow definitions that handle aggregation modules (Section 4.3.2.2 on page 55). Investigating modules in $\mathrm{DescAggregations}$ suffices because it will include all modules that are potentially applicable for the current layer, since the algorithm performs a greedy search that will add all applicable aggregations and since the set $\mathrm{DescModules}$ increases monotonic for layers closer to the root. If an aggregation is applicable, the algorithm adds it to $\mathrm{Modules}(l)$ and to $\mathrm{AggregationForwards}(l, l')$. At the end of the first step, the algorithm added all modules directly mapped onto the current layer to $\mathrm{Modules}(l)$, as well

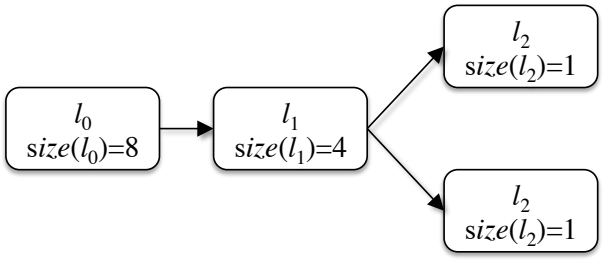


Figure 4.12: *An example layer tree for a layout with two root places allows an illustration of the aggregation placement decisions of the algorithm in Figure 4.11.*

	l_0	l_1	l_2	l_3
$m_{L,M}$	$\emptyset$	$\emptyset$	$\{m_2\}$	$\{m_0\}$
Step 1 adds	$\emptyset$	$\{m_1\}$	$\emptyset$	$\emptyset$
Step 2 adds	$\emptyset$	$\emptyset$	$\emptyset$	$\{m_1\}$
DescModules	$\{m_2, m_0\}$	to l_2: $\{m_2\}$ to l_3: $\{m_0\}$	$\emptyset$	$\emptyset$
DescAggregations	$\{m_1\}$	to l_2: $\emptyset$ to l_3: $\{m_1\}$	$\emptyset$	$\emptyset$
AggregationForwards	$\emptyset$	to l_2: $\emptyset$ to l_3: $\{m_1\}$	$\emptyset$	$\emptyset$
Modules	$\emptyset$	$\{m_1\}$	$\{m_2\}$	$\{m_0, m_1\}$

Table 4.1: *Illustration of module placement algorithm (Figure 4.11) results and variables for the layer tree in Figure 4.12.*

as any aggregation module that supports a module of a successor layer l', and that is applicable for all modules on successor layers of l'.

The second step, lines 13–14, of the algorithm extends the set $\text{Modules}(l)$ for the current layer l such that it includes all dependencies of any module in the set, as well as any supported aggregation for a module in this set. The resulting set $\text{Modules}(l)$ can violate the *Aggregation-Applicability* rule (Event-Flow 6). Thus, the third step of the algorithm, removes all aggregation modules from $\text{Modules}(l)$ that violate the *Aggregation-Applicability* condition. As a result, for each layer l, the algorithm yields a set $\text{Modules}(l)$ that satisfies the event-flow definitions for aggregations, as well as a set $\text{AggregationForwards}$ that highlight for which direct successors a layer can forward aggregated events of which aggregation module.

Consider the introductory tool specification in Figure 4.3 on page 40 as an example to illustrate the module placement algorithm. The algorithm computes: $\text{Modules}(l_0) = \{m_0\}$, $\text{Modules}(l_1) = \emptyset$, and $\text{Modules}(l_2) = \{m_0, m_1\}$. Lines 13–14 compute the full output in this example, since it only uses a single module dependency and no aggregations. Thus, lines 9–12 will not add any aggregation modules, while lines 15–18 will not remove any incompatible modules.

Figures 4.7(b) and 4.7(d) on page 54 illustrate a second example. It uses the analysis module m_0 that supports the aggregation module m_1. To illustrate the impact of the layer-based aggregation module decisions in the first step of the algorithm, consider an additional module m_2 that is equal to the version of m_0 in Figure 4.7(a), i.e., m_2 uses an analysis that is mapped to h_0 and does not support aggregation m_1. Figure 4.12 presents a layer tree to instantiate a tool with these modules and Table 4.1 presents the module mapping ($m_{L,M}$) for this example. The table also illustrates the results and intermediate variables of the module placement algorithm in Figure 4.11. Layers l_3 and l_2 have no descendants in the layer tree. Thus, step 1 adds no modules for these layers. Step 2 then adds dependent modules (none), as well as aggregation modules for layers l_3 and l_2 (m_1 on l_3). For layer l_1, the first step of the algorithm determines that the aggregation m_1 is applicable for the direct successor layer l_3, but not for l_2. This

results from the fact that m_0 on layer l_3 supports the aggregation module, while m_2 on layer l_2 does not. Steps 2 and 3 of the algorithm then neither add nor remove any modules for layer l_1. For layer l_0 the algorithm determines that the aggregation m_1 is not applicable since the descendant module m_2 does not support this aggregation, while both modules have a mapping to h_0, i.e., $(m_1, \{m_2, m_0\}) \notin \mathrm{aggregationCompatible}$.

After module placement, a consistency check can warn whether intralayer communication might not be available on all layers that want to employ it. This holds if a layer $l \notin L_{\mathrm{intra}}$ executes a module $m \in \mathrm{Modules}(l)$ that is mapped to a hook h with $\mathrm{dir}(h) = \mathrm{intralayer}$. The weaver implementation of GTI warns tool developers that such modules may not function correctly. At runtime, if a module on a layer without intralayer communication wants to inject an intralayer event, the GTI tool infrastructure implementation will not provide a function pointer for event injection. Such modules may then use an alternative implementation, e.g., primary and broadcast communication direction in combination, or abort.

4.4.2 Analysis Input Forwarding

The event-flow definitions of the previous sections detailed the actions that a place must execute when it observes a hook or when it receives an event from a hook. However, these definitions do not detail the information that a tool infrastructure implementation must communicate at runtime. As to reduce tool overheads, events should not include any superfluous information. Thus, GTI's weaver—after it placed modules onto layers—shapes the contents of events for all hooks, i.e., it ensures that events only include necessary data fields.

The event-flow definitions define which hooks can create events. Then, the tool layout, the module placement, and the communication directions of the individual hooks provide the necessary information to shape events. As an example, if the root layer—containing application/leaf places—uses no modules, while successive layers use modules: the places of the root layer must still observe all hooks that use the primary communication direction and to which a successor layer module is mapped. The GTI implementation uses *event shapes* to define the event contents that wrapper modules create and that arrival modules forward to other places. Particularly, the weaver determines *minimal* contents for the events that it creates. As an example, if a hook provides an argument that no analysis of a mapped module uses, then no event needs to include this argument. In addition, when events progress through places, parts of their information may become unnecessary at certain layers. GTI's weaver provides event shapes for each layer, such that arrival modules can reshape events to only include required information. Thus, *minimal events* refers to events that only carry information on arguments and operation results that some module analysis on the remaining part of the event's communication direction requires. However, this includes no encoding or redundancy detection techniques, e.g., as opposed to byte encoding in an OTF2 extension [166].

Analysis-hook mappings (Section 4.2.6 on page 47) use sequences to specify which operations or hook arguments form the individual inputs of an analysis. In the following, the term *input* refers to both hook arguments and the results of operations for brevity. Events must contain all inputs that any analysis on the event's communication path requires. However, where the mapping requires the use of sequences that specify the input order for each analysis, to compute event shapes, the weaver only requires information on the inputs that an event must contain. Thus, the weaver represents event shapes as sets of inputs. Particularly, a representation with sets removes duplicate inputs in the subsequent algorithms. The function $\mathrm{setify} : \bigcup_{i,j\in\mathbb{N}} \mathrm{ASeqs}_{i,j} \to \{i | i \in \mathbb{N} \vee i = (o \in O, seq \in \bigcup_{i,j\in\mathbb{N}} \mathrm{Seqs}_{i,j}\}$ translates mapping sequences into sets, with:

$$\mathrm{setify}\big((i_0, i_1, \ldots)\big) = \{i_0, i_1, \ldots\}$$

Figure 4.13 illustrates an algorithm to calculate event shapes. It calculates sets $\mathrm{RequiredInputs}(l, h) \in \mathrm{codomain}(\mathrm{setify})$ that provide the event contents that layer l must perceive for hook h. The algorithm calculates event contents for the primary and the intralayer communication directions, while a second

Function `calculateRequiredInputs(`l`)`

```
   /* Initialize all sets of required inputs as empty                    */
1  for h ∈ H do
2      RequiredInputs(l, h) := ∅
   /* Add own used inputs                                                */
3  for m ∈ Modules(l) do
4      for a ∈ a_M(m) do
5          for h ∈ H with (a, h) ∈ AH do
6              RequiredInputs(l, h) := RequiredInputs(l, h) ∪ setify(m_{A,H}(a, h))
   /* Add inputs from layer tree descendants                             */
7  for l' with (l, l') ∈ E_L do
8      calculateRequiredInputs(l)
9      for h ∈ H with dir(h) = primary do
10         RequiredInputs(l, h) := RequiredInputs(l, h) ∪ RequiredInputs(l', h)
```

Figure 4.13: *In order to compute the shape of events and to determine event forwarding, the illustrated algorithm computes sets of inputs that each layer must observe/receive for each hook.*

Function `forwardInputsForBroadcast(`l`)`

```
   /* Add inputs to layer tree descendants                               */
1  for l' with (l, l') ∈ E_L do
2      for h ∈ H with dir(h) = broadcast do
3          RequiredInputs(l', h) := RequiredInputs(l', h) ∪ RequiredInputs(l, h)
4      forwardInputsForBroadcast(l)
```

Figure 4.14: *An algorithm that complements the algorithm in Figure 4.13 by computing sets of inputs that layers must observe or receive for each hook along the* broadcast *direction.*

algorithm then completes the event shape for the broadcast direction. The weaver passes the root layer as the input argument to the algorithm in Figure 4.13 and a recursion then handles all remaining layers. Lines 1–2 initialize the result sets for all hooks with an empty set. Afterwards, for each hook, lines 3–6 add all inputs that analyses of modules on the current layer require. For hooks that use the intralayer direction, this set contains the required minimal information, since these hooks originate and target the same layer. Lines 7–10 of the algorithm first issue a recursion into all descendant layers. Afterwards, for the primary communication direction, the current layer must perceive any input that a successor layer requires. Thus, the algorithm adds required inputs of all direct successor layers for hooks that use the primary communication direction.

Table 4.2 presents results and algorithm variables for the algorithm in Figure 4.13. The introductory example tool from Figure 4.3 (page 40) serves as the input for the algorithm. The first row of the table summarizes the module placement result from the last section. Afterwards, the table provides the input sets that the modules of each layer use (lines 3–6 of the algorithm). Finally, the RequiredInputs row summarizes the result of the algorithm for each layer and hook, which results from adding successor layer inputs (lines 7–10).

For hooks that use the broadcast communication direction, a layer must provide all inputs that any predecessor layer in the layer tree uses. Figure 4.14 illustrates a second algorithm that the weaver issues after the algorithm in Figure 4.13. The root layer again serves as the initial input that the weaver provides to the algorithm. A recursion then issues the forwarding for all other layers. The algorithm inverts the propagation of the required inputs and only applies to hooks of the broadcast communication direction.

		l_0	l_1	l_2
Modules		$\{m_0\}$	$\emptyset$	$\{m_0, m_1\}$
Lines 3–6	h_0	$\{0\}$	$\emptyset$	$\{0\}$
	h_1	$\emptyset$	$\emptyset$	$\{0, (o_0, (0,1)), 1\}$
RequiredInputs	h_0	$\{0\}$	$\{0\}$	$\{0\}$
	h_1	$\{0, (o_0, (0,1)), 1\}$	$\{0, (o_0, (0,1)), 1\}$	$\{0, (o_0, (0,1)), 1\}$

Table 4.2: *An illustration of intermediate and final results of the algorithm in Figure 4.13 for the example from Figure 4.3.*

4.5 Order Preserving Aggregation

Event order can influence the results of analyses. As an example, the MPI standard requires that derived MPI datatypes in communication operations are *committed*. A module that checks whether an application violates this restriction would be mapped to communication operations, e.g., `MPI_Send`, and to operations that introduce/commit derived datatype, i.e., `MPI_Type_commit`. When the module perceives a communication operation, it checks whether the given datatype is derived, and if so, whether it was committed beforehand. The analysis can fail to report defects, or can report false positives, if event processing on tool places would modify event order, e.g., a commit event that originally preceded a communication operation could be passed to the analysis only after it analyzed the communication operation. This section presents a concept that preserves process local event order in the presence of aggregations.

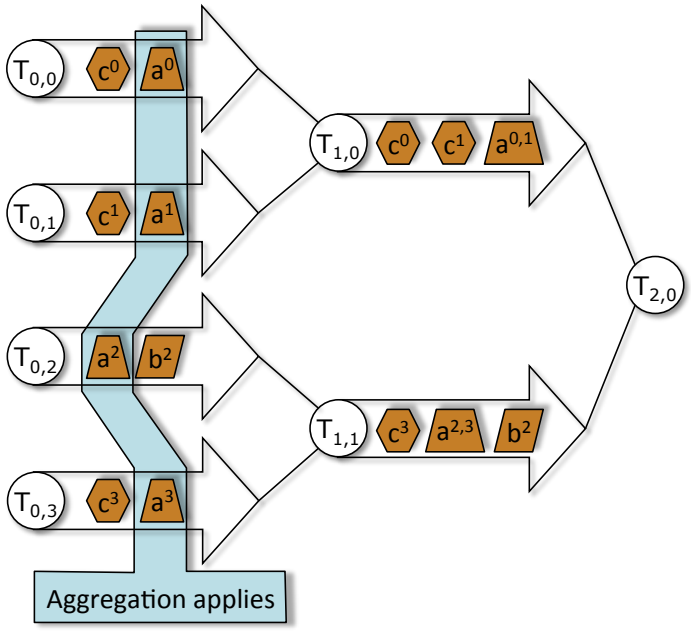
4.5.1 Order

Timestamping techniques use logical timestamps [103], vector clocks [44, 112], and further structures to determine event order. Usually these approaches include communication calls in their relation, such that, if process i sends process j a message, then any event of i that occurred before its send event is ordered before any event of j that occurs after its receive event. However, since the MPI standard leaves MPI implementations freedoms as to whether collective operations are synchronizing and whether standard mode send operations are blocking, such a relation needs to adapt to choices of the MPI implementation to cover actual event order. As a result, if a tool infrastructure for the proposed abstraction is required to consider event order across multiple processes, then it would both need to consider information on the semantics of the underlying parallel programming paradigm and on runtime choices of the implementation of the paradigm. As a result, the proposed abstraction does not require an enforcement of event order across processes, as to avoid these demands. Thus, the proposed abstraction only considers event order between events that originate from the same process and uses the term *event order* to refer to this process local order for brevity.

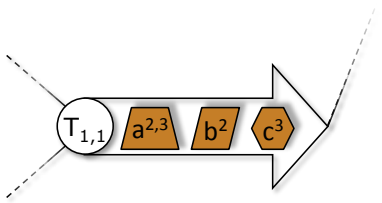
For the primary communication direction, process local order requires that if process i creates an event e before e', then all analyses (on any place) must process e before e'. Tool developers can derive such an order manually if they associate timestamps with events. However, this imposes additional complexity on tool modules, while in addition, the discussion below highlights that aggregations complicate timestamping mechanisms. Thus, GTI provides algorithms that automatically preserve event order in the presence of aggregations.

Tool places enforce event order with a buffering approach that uses the channelTreeQueues data structure. The previous place driver algorithm from Figure 4.10 (page 62) illustrates how the main loop of a tool place interacts with this data structure. Without aggregations, this data structure satisfies:

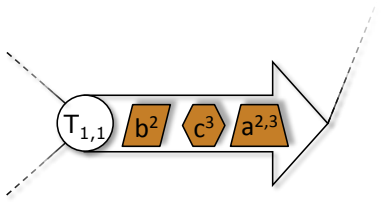
- channelTreeQueues.findEventToDequeue always returns $(\bot, \text{NULL}, \epsilon)$,
- channelTreeQueues.canProcess always returns $\top$, and
- arrival.processPrimaryDirectionEvent always returns $(\bot, ())$.



(a) Four places $T_{0,0}$–$T_{0,3}$ create a mix of events. An aggregation applies to a_0–a_3. Places $T_{1,0}$ and $T_{1,1}$ illustrate an event forwarding that preserves process local event order.



(b) An incorrect output stream of $T_{1,1}$ that violates process local event order for event c^3.



(c) A second incorrect output stream of $T_{1,1}$ that violates process local event order for event b^2.

Figure 4.15: *An input event stream highlights that order preserving event aggregation must consider the state of ongoing aggregations.*

The main loop algorithm immediately preserves event order with these return values, i.e., without aggregations. This follows from the observation that leaf places forward events in their order and that all tool places process events in the order in which they receive them. Thus, without event aggregation, all places process and forward events in their order.

Figure 4.15 presents a tool layout with four application places and two layers of tool places. The illustration uses the event stream representation from Section 3.4.1 (page 29) to highlight an event mix that uses the primary communication direction. This event mix includes events of three types: a (trapezium shape), b (trapezoid shape), and c (hexagon shape). The superscripts of the events specify the leaf place identifiers (0–3) of the leaves that contributed information to them. As an example, event a^0 is an event of type a that place $T_{0,0}$ (application process 0) created, whereas an event $a^{0,1}$ is an aggregated event of type a that replaces events from places $T_{0,0}$ and $T_{0,1}$, i.e., contains information from these two leaves. In Figure 4.15, an event aggregation applies to all events of type a, and $T_{1,0}$, $T_{1,1}$, and $T_{2,0}$ apply the respective aggregation module. Especially, the processing on place $T_{1,1}$ impacts event order. This place could receive the events from $T_{0,2}$ and $T_{0,3}$ in the following order:

Event a^3 first, b^2 second, c^3 third, and a^2 as the last event.

Since the aggregation on $T_{1,1}$ uses a^2 and a^3 as its inputs to create an aggregated event $a^{2,3}$, the place must handle the two intermediate events b^2 and c^3.

A first approach could continue event processing when aggregations start. With that, when $T_{1,1}$ handled a^3 that starts the aggregation (waiting for a^2 to complete), it will continue to process and forward events. Thus, $T_{1,1}$ processes and forwards events b^2 and c^3, before it finally processes a^2, which finishes the aggregation and creates the aggregated event $a^{2,3}$. In summary, the node forwards events in the order that Figure 4.15(b) illustrates. Particularly, the aggregated event $a^{2,3}$ is the last event that $T_{1,1}$ forwards. This output loses event order since: $T_{1,1}$ forwards c^3 before $a^{2,3}$, while a^3 preceded c^3 originally (the aggregated event contains the information of a^3).

A second approach could delay event processing/forwarding until aggregations complete. In that case, when $T_{1,1}$ processes a^3 and starts its aggregation, the place would buffer events b^2 and c^3. When the

place receives and processes event a^2, it creates and forwards the aggregated event $a^{2,3}$. Afterwards, it would process and forward all queued events, which leads to the output event stream in Figure 4.15(c). This output again violates event order since: $T_{1,1}$ forwards $a^{2,3}$ before b^2, while originally b^2 preceded a^2.

Thus, tool places must use an aggregation handling that buffers events such that event processing and forwarding preserves event order. The following presents the algorithms and techniques that this thesis provides for this purpose. The GTI prototype implements this approach.

4.5.2 Channel Identifiers

A set of leaf place indices can express which leaf places provided information to an event. A buffering algorithm [74] can ensure that each place preserves event order if GTI associates such a set of indices with each event. For an event e, the list $\text{indices}(e)$ specifies the leaf place indices of all leaf places that contributed information to e. If a leaf place $T_{0,i}$ creates an event e, then $\text{indices}(e) \stackrel{def}{=} \{i\}$. While, if an aggregation replaces events $e_0, e_1, \ldots, e_k$ with an event e, then $\text{indices}(e) \stackrel{def}{=} \bigcup_{x=0}^{k} \text{indices}(e_x)$. Finally, if a tool place T injects an event e, then $\text{indices}(e) \stackrel{def}{=} \{i : T_{0,i} \text{ is predecessor of } T \text{ in the topology graph}\}$. On each tool place, a buffering algorithm [74] can keep a leaf place state list that supports decisions on whether processing an event in the presence of active aggregations violates event order or not. In addition, each place manages a queue of events that it cannot process at the moment. The leaf states mark a set of leaf place indices I that includes all leaf place identifiers that provided information to an event that an active aggregation uses. I is the smallest set that satisfies: If an active aggregation uses events $e_0, e_1 \ldots, e_k$ as inputs, then $\bigcup_{x=0}^{k} \text{indices}(e_x) \subseteq I$.

A tool place with an ongoing aggregation leaf identifier set I and an event queue $e_0, e_1, \ldots, e_k$ can process a newly received event e (of the primary communication direction) if:

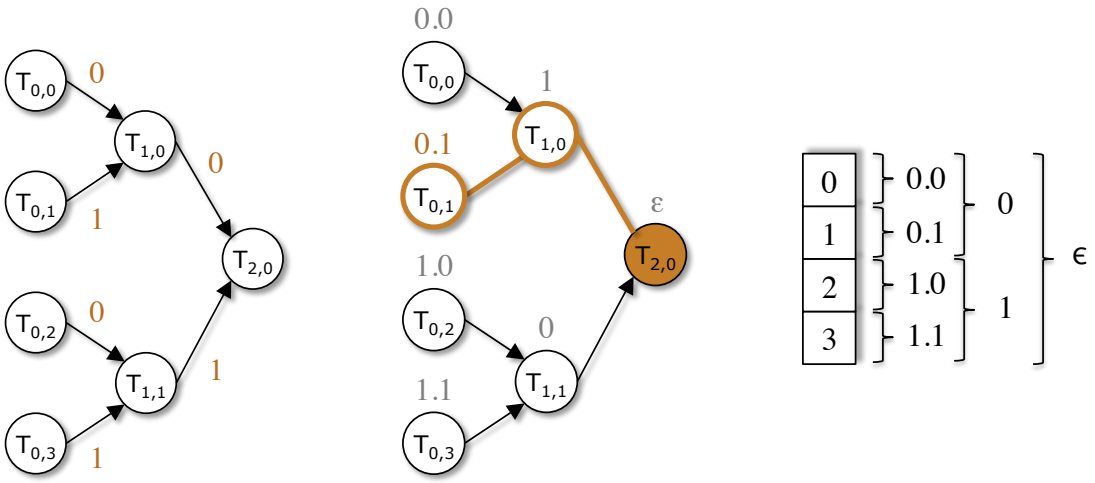
$$\bullet\ I \cap \text{indices}(e) = \emptyset, \text{ and} \tag{4.4}$$

$$\bullet\ \forall x \in \frac{1}{2}\{1, \ldots, k\} : \text{indices}(e_x) \cap \text{indices}(e) = \emptyset. \tag{4.5}$$

Condition (4.4) specifies that a place must not process an event e if it shares a leaf place identifier with an event e' that an active aggregation uses. Otherwise, e would overtake e' while e arrived after e'. In addition, condition (4.5) ensures that e will not overtake any event that was queued beforehand and shares a leaf place identifier with e.

Tool places can evaluate these two conditions if each event e in the communication system carries information on $\text{indices}(e)$ as a list. Assume that p leaf places each create one event, for which an aggregation reduces this wave towards a single event on the root of a tree layout. In this scenario, each successive aggregation increases the number of leaf place indices, such that the root receives p indices in total. Thus, even with an efficient aggregation, the root place would require $\mathcal{O}(p)$ time and memory to receive and evaluate these indices. Thus, the use of leaf place index sets limits tool scalability.

The subsequent algorithm for order preserving event aggregation uses *channel identifiers*, which represent paths in a tree, to overcome this limitation. On each tool place, a *channel tree* defines a directed rooted tree that is a subtree of the topology graph. The channel tree uses the node that represents the tool place in the topology graph as root. It then uses all predecessors of the tool place node in the topology graph as its node set. The edges of the channel tree then restrict the edges of the topology graph to the node set of the channel tree. Figure 4.16(a) presents this arborescence for a place $T_{2,0}$, e.g., the root place from the layout in Figure 4.15. Let a channel index be associated with each arc of the channel tree, such as Figure 4.16(a) illustrates. Then, *channel identifiers* represent path expressions with channel indices. These identifiers start from the root of a channel tree and point to a single node of the tree. Figure 4.16(b) highlights the channel identifier 0.1 for the channel tree of node $T_{2,0}$, which evaluates to node $T_{0,1}$ (highlighted in the figure). Particularly, channel identifiers are evaluated from left to right to construct their path. Thus, to evaluate the identifier 0.1: First use channel 0 of the channel tree root to arrive at $T_{1,0}$, then use channel 1 from this node to arrive at node $T_{0,1}$. As a second example, the channel



(a) The channel indices (0 and 1 in the example) identify input channels of places.

(b) All channel identifiers for place $T_{2,0}$ and a highlighted path for the identifier 0.1.

(c) Channel identifiers represent continuous ranges of leaf place indices. The illustration highlights these ranges for all possible channel identifiers on place $T_{2,0}$.

Figure 4.16: *An illustration of a channel tree and channel identifiers for place $T_{2,0}$.*

identifier 1 evaluates to node $T_{1,1}$. For order preserving event aggregation, a channel identifier must be associated with each event, in order to store which leaf places provided information to the event. As an example, an event with channel identifier 0.1 on $T_{2,0}$—Figure 4.16(b)—specifies that only the leaf place $T_{0,1}$ provided information to this event.

Figure 4.16(c) relates channel identifiers with sets of leaf place indices: For place $T_{2,0}$, the channel identifier 0.1 represents the set of leaf place indices $\{1\}$, whereas the channel identifier 1 represents the set $\{2, 3\}$. Thus, channel identifiers can represent sets of leaf place indices. However, while leaf place identifier sets can adept to aggregations that only use events from some—but not all—leaf places, channel identifiers use a hierarchical coarsening that will superset such sets. In the example, a leaf place set such as $\{0, 2\}$ requires the use of the channel identifier ϵ—empty path—as a superset.

GTI stores channel identifiers in 64 bit values. For a k-ary tree layout, each component of a channel identifier requires $\mathrm{ld}(k + 1)$ bits, of which k states represent the channel index and the $(k + 1)$-th state represents whether a channel identifier uses a certain component or not, e.g., to distinguish the channel identifier 1 from the identifier 1.0. As an example, for a binary tree layout, each channel identifier component requires 2 bits. Thus, a single 64 bit value stores 32 components that can represent channel identifiers in a tree layout with 2^{32} leaves. Thus, channel identifiers render the $O(n)$ memory requirement for leaf place index sets into an $O(\mathrm{ld}\, n)$ requirement that enables scalability. This improvement comes at the cost that channel identifiers cannot represent all leaf place index sets. The over-approximation that channel identifiers use to represent such sets then impacts the subsequent algorithms.

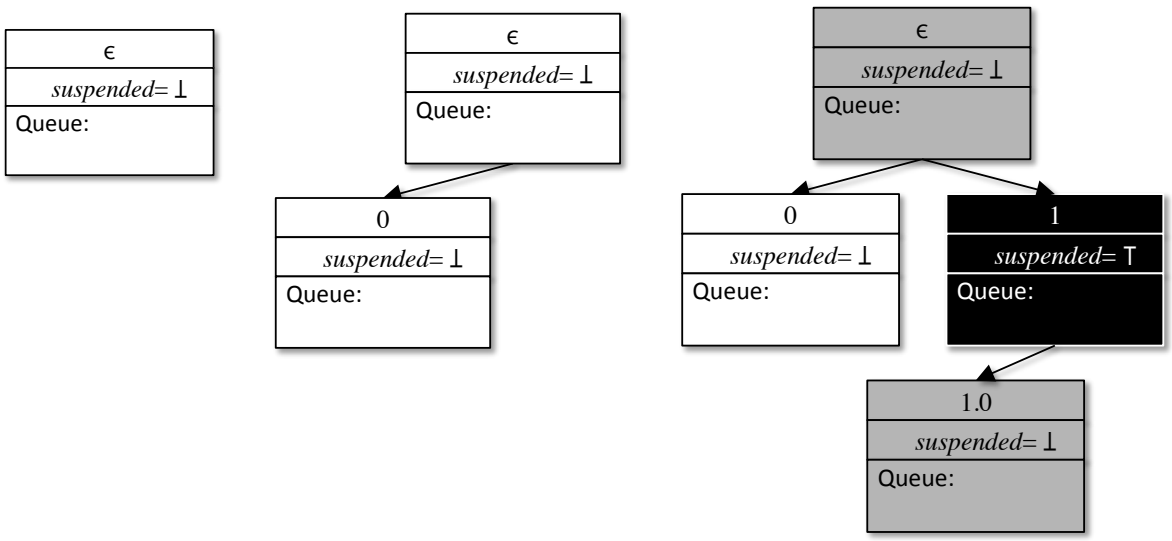
4.5.3 Tree Queue Algorithms

A *queue tree* combines state information and event queues with channel trees to enable algorithms for order preserving event aggregation. Queues in the queue tree hold events that a place could not process, as to not violate event order. GTI's data structure for nodes of this tree contains the following fields:

childs**:** Pointers to child nodes,

suspended**:** Boolean state, where $\top$ represents that an active aggregation uses an event whose channel identifier corresponds to this node, and

queue**:** A queue of events that could not be processed previously.



(a) An initial queue tree that consists of only a root node.

(b) A queue tree that results after adding a node for the channel identifier 0.

(c) A queue tree that results after adding nodes for the channel identifier 1.0. For subsequent illustration purposes the node for the identifier 1 is marked as suspended (box with white text on black background).

Figure 4.17: *Examples of queue trees on a tool place during event processing. The illustration highlights the data structures and data fields.*

Note that a channel tree of a place T is directed towards the node that represents T, since it is based on a subgraph of the topology graph, e.g., arc directions in Figure 4.16(a). The following considers a parent-child relationship for queue trees that inverts the arc directions of channel trees. Subsequently, the terms *descendant* and *ancestor* refer to this parent-child relationship: T' is a descendant of T if T' is a child of T; and If T'' is a descendant of T' and T' is a child of T, then T'' is a descendant of T. The arc directions for the queue trees in Figure 4.17 illustrate this relation.

Each tool place initiates the queue tree with a root that it associates with the channel identifier ϵ. Figure 4.17(a) illustrates this root node along with its default data fields. During event processing, places execute algorithms that operate on its local queue tree. These algorithms add new child nodes that represent specific channel identifiers to store state information. As a result, the queue tree on tool place T represents a subset of the channel tree of T. Figure 4.17(b) illustrates a queue tree with a root node and a node that represents the channel identifier 0, for a place such as $T_{2,0}$ of the layout in Figure 4.15. A further query to a queue tree algorithm could add a node for the channel identifier 1.0, which adds an intermediate node for the channel identifier 1 in the subsequent algorithms. Figure 4.17(c) illustrates the structure of the resulting queue tree. The figure assigns $\text{suspended} = \top$ for the intermediate node of channel identifier 1 to illustrate subsequent concepts.

Algorithms that preserve event order in the presence of aggregations need to evaluate conditions (4.4) and (4.5), which test whether sets of leaf place indices overlap. With queue trees, and the leaf place indices that channel identifiers represent, two identifiers cid_1 and cid_2 overlap exactly if (without proof):

- $\text{cid}_1 = \text{cid}_2$,
- cid_1 is an ancestor of cid_2 in the queue tree, or
- cid_2 is an ancestor of cid_1 in the queue tree.

Figure 4.17(c) uses the queue tree node for channel identifier 1 to illustrate these conditions. It marks that an active aggregation uses an event with this channel identifier, i.e., that the node is *suspended* from processing (node with black background). All leaf place indices of nodes with gray or black background

Function `suspendChannel(cid)`

```
1  if cid = ϵ then
2  |  this.suspended := ⊤
   |  /* ...update descendantSuspendCount of all ancestors ...  */
3  else
4  |  this.getChildForChannelIdentifier(cid).suspendChannel(pop(cid))
```

Function `openChannel(cid)`

```
1  if cid = ϵ then
2  |  this.suspended := ⊥
   |  /* ...update descendantSuspendCount of all ancestors ...  */
3  else
4  |  this.getChildForChannelIdentifier(cid).openChannel(pop(cid))
```

Function `canProcess(cid)`

```
   /* This node overlaps with cid if this node has a non-emtpy queue
      or is suspended; if so return ⊥                            */
1  if this.suspended = ⊤ ∨ this.queue.empty() = ⊥ then
2  |  return ⊥
   /* Recursion to the node that represents cid                  */
3  if cid = ϵ then
   |  /* This node represents cid, check if descendants are suspended
   |     or have non-empty queues                                */
4  |  if this.descendantQueueSize > 0 ∨ this.descendantSuspendCount > 0 then
5  |  |  return ⊥
6  |  else
7  |  |  return ⊤
8  else
   |  /* Continue recursion                                      */
9  |  child := this.getChildForChannelIdentifier(cid)
10 |  return child.canProcess(pop(cid))
```

Figure 4.18: *Algorithms on queue trees to suspend/open tree nodes and to determine whether events with given channel identifiers can currently be processed.*

overlap with the leaf place indices that the channel identifier 1 represents.

Tool places use the subsequent algorithms to ensure that if an active aggregation suspends a node in the queue tree, then the place does not process events with overlapping leaf place indices, i.e., the algorithms implement condition (4.4). The queue tree algorithms also ensure that if a node contains events in its queue—that could not be processed due to (previously) active aggregations—then events with channel identifiers that specify overlapping leaf place indices will also not be processed, i.e., the algorithms implement condition (4.5). A place enqueues an event in one of the queue tree nodes if it cannot be processed due to either of the two conditions.

Figures 4.18 and 4.19 present the queue tree algorithms that suspend nodes (suspendChannel), remove suspensions of nodes (openChannel), specify whether a place can process an event (canProcess),

Function `enqueue` (e, cid)

```
1  if cid = ε ∨ this.queue.empty() = ⊥ then
2      this.queue.push((e, cid))
       /* ...update descendantQueueSize of all ancestors ...        */
3      return
4  this.getChildForChannelIdentifier(cid).enqueue(e, pop(cid))
```

Function `findEventToDequeue` (cid)

```
   /* If this node is suspended, no events can be dequeued in it or
      any of its childs                                              */
1  if this.suspended = ⊤ then
2      return (⊥, NULL, cid)
   /* Prune the search if there are no events to be found            */
3  if this.queue.empty() ∧ this.descendantQueueSize = 0 then
4      return (⊥, NULL, cid)
   /* Search in childs first, their events preceed this nodes events
      */
5  for (child, channel) ∈ this.childs do
6      (success, e, cidOfE) := child.findEventToDequeue(concat(cid, channel))
7      if success = ⊤ then
8          return (⊤, e, cidOfE)
   /* If no descendant has an event to dequeue, look into our queue
      (if no child is suspended)                                     */
9  if this.descendantSuspendCount = 0 ∧ this.queue.empty() = ⊥ then
10     (e, cidOfE) := this.queue.pop()
       /* ...update descendantQueueSize of all ancestors ...        */
       /* Push events that belong to descendants downwards          */
11     while this.queue.empty() = ⊥ do
12         (g, cIdOfG) := this.queue.front()
13         if cIdOfG = ε then
14             break
15         else
16             this.queue.pop()
               /* ...update descendantQueueSize of all ancestors ...*/
17             this.getChildForChannelIdentifier(cIdOfG).enqueue(g, pop(cIdOfG))
       /* Return the front of this nodes queue                      */
18     return (⊤, e, concat(cid, cidOfE))
19 return (⊥, NULL, cid)
```

Figure 4.19: *Algorithms on queue trees to enqueue events and to find and dequeue events that a layout node can process after it completes an aggregation.*

enqueue an event (enqueue), and dequeue an event (findEventToDequeue). The algorithms represent queue tree nodes as objects that use the fields childs, suspended, and queue. Additionally, the algorithms use "this." to refer to these fields of a current node. Some of the queue tree algorithms need to determine whether a descendant node is suspended or has a non-empty queue. To avoid repeated searches for these properties, each node carries two additional fields descendantSuspendCount and descendantQueueSize.

Tool places use the suspendChannel (Figure 4.18) algorithm to suspend a node for a channel identifier once an active aggregation uses an event with this identifier as input. Additionally, tool places use the openChannel (Figure 4.18) algorithm to remove a suspension from a node after the aggregation that suspended the node completes. Both algorithms traverse the tree until they find the node that represents their input channel identifier. To do so, they utilize the function getChildForChannelIdentifier that reads the leftmost component of a channel identifier, determines whether a child node exists for this component, and either returns the existing child or creates a new one. In addition, the utility function pop(cid) removes the leftmost component from a channel identifier to trim channel identifiers for recursion. Both algorithms sketch the update of the descendantSuspendCount field of their ancestors, where the GTI implementation utilizes an additional parent pointer on each node.

The canProcess algorithm (Figure 4.18) implements conditions (4.4) and (4.5) to determine whether an event may be processed or not. The algorithm uses recursion to arrive at the node that represents the channel identifier of the event in question. If during this recursion, an intermediate node has a non-empty queue or is suspended, then the algorithm immediately returns $\bot$. Once the algorithm arrives at the target node, the algorithm checks whether a child node is suspended or has a non-empty queue, in which case it returns $\bot$. Otherwise, it returns $\top$.

The enqueue algorithm (Figure 4.19) traverses towards the node that represents the channel identifier of the input event and adds it to the queue of this node. However, if an ancestor of the target node has a non-empty queue, the algorithm adds the event to the first such ancestor instead. This handling is necessary since the queue tree algorithms dequeue events of descendant nodes before events of their ancestors. Thus, if enqueue would step over nodes with non-empty queues during its recursion, it could violate event order. Queues of queue tree nodes store both the actual event and the remaining channel identifier that the enqueue algorithm uses for its recursion. The findEventToDequeue algorithm then uses the channel identifier reminder to push events into their actual target nodes at a later time.

The findEventToDequeue algorithm (Figure 4.19) determines whether a queue tree node for a channel identifier cid has a non-empty queue and whether it satisfies $\text{canProcess}(\text{cid}) = \top$. It returns a triple $(\text{success}, e, \text{cid})$, where success specifies whether the algorithm found an event to dequeue. If the algorithm finds an event to dequeue it returns the event in e and its associated channel identifier in cid. The first two if-conditions of the algorithm prune the search if the current node is suspended or if no events are queued in any descendant. Afterwards, the algorithm uses recursion to investigate the child nodes of the current node first. If a descendant is successful, the algorithm returns the event that the recursive call returned. If no child finds a suitable event, the algorithm checks whether the queue of the current node is non-empty and whether no child node is suspended as well. If so, the algorithm removes the front element of the current node's queue and returns this event. In addition, the algorithm redistributes queued events if they had previously been queued to an ancestor of their target node.

Appendix A.2 (page 164) presents the queue tree algorithm queries that the main driver algorithm of a tool place creates for the input events that $T_{2,0}$ receives in Figure 4.15(a). Additionally, it visualizes the individual states of the queue tree on $T_{2,0}$ to detail the previous algorithms.

4.5.4 Time Complexities

Table 4.3 summarizes the time complexities of the algorithms in Figures 4.18 and 4.19, as well as of their utility functions. The "worst case" column specifies worst case complexities for cases where aggregations do not apply or fail, i.e., where events that originate from leaf places reach root places. The "Worst case, successful aggregations apply" column specifies worst case complexities for situations in

	Layout: k-ary tree, p leaves, l queue entries	
	Worst case	Worst case, successful aggregations apply
Utility:		
getChildForChannelIdentifier	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Update ancestors descendantQueueSize	$\mathcal{O}(\log_k p)$	$\mathcal{O}(1)$
Update ancestors descendantSuspendCount	$\mathcal{O}(\log_k p)$	$\mathcal{O}(1)$
Queries:		
suspendChannel	$\mathcal{O}(\log_k p)$	$\mathcal{O}(1)$
openChannel	$\mathcal{O}(\log_k p)$	$\mathcal{O}(1)$
canProcess	$\mathcal{O}(\log_k p)$	$\mathcal{O}(1)$
findEventToDequeue	$\mathcal{O}(p + l \log_k p)$	$\mathcal{O}(k)$
enqueue	$\mathcal{O}(\log_k p)$	$\mathcal{O}(1)$

Table 4.3: *Time complexities for queue tree algorithms and their utility functions. Column "Worst case" represents a worst case situation in which aggregations may not apply or fail. Column "Worst case, successful aggregations apply" represents a worst case situation in which aggregations apply to all events and always succeed.*

which aggregations apply to all events and always succeed, i.e., if all places only receive events of the primary communication direction that were created by their direct predecessors in the layout. Without successful aggregations (first column), all algorithms, except for findEventToDequeue, traverse the tree towards a node that represents an input channel identifier, which results in a time complexity of $\mathcal{O}(\log_k p)$ for a k-ary tree layout with p leaves. In addition, these algorithms may need to update suspension or queue counts on ancestor nodes, which requires the same cost. The $\mathcal{O}(\log_k p)$ cost adds a processing time to each event that increases with application scale, but the logarithmic increase does not prohibit scalability immediately. However, the algorithm findEventToDequeue can limit scalability with an $\mathcal{O}(p + l \log_k p)$ time complexity. In worst case most leaf nodes of a queue tree are suspended and have non-empty queues, which requires the algorithm to explore $\mathcal{O}(p)$ leaves. If in addition, the last child of the root node has l queued events, no suspension, and no descendant with a non-empty queue or suspension; then the algorithm may in addition have to re-queue $l - 1$ events. However, in this worst case scenario, the root receives $\mathcal{O}(p)$ events in total, i.e., is a scalability bottleneck in the first place.

The time complexities with successful aggregations—second column—are usually $\mathcal{O}(1)$ and represent costs for scalable tools that employ effective event aggregations. The time complexity of the findEventToDequeue algorithm is $\mathcal{O}(k)$ in this scenario, since all places will use queue trees with at most k children. This cost provides a small overhead for order preserving event aggregation, but does not limit scalability.

4.5.5 Applicability

Channel identifiers represent leaf place index sets with memory requirements that are logarithmic in the number of application processes. Additionally, the queue tree algorithms require logarithmic or lower time complexities (If aggregations apply and are successful). However, channel identifiers cannot represent some leaf place index sets, which requires that a tool uses channel identifiers that supersets them. Figure 4.20 illustrates a scenario with two event types a and c. Leaf places of an even index create an event of type a, while leaf places with an odd index create an event of type c. An aggregation applies to waves of each event type respectively. The aggregations on nodes $T_{1,0}$ and $T_{1,1}$ replace single events with a new aggregated event, e.g., $a^{0,1}$. As a result, $T_{2,0}$ receives two events with channel identifier 0 from $T_{1,0}$ and two events with channel identifier 1 from $T_{1,1}$. If the root processes $a^{0,1}$ first and then $c^{2,3}$, its queue tree will not allow any further event processing. However, since neither the aggregation for events of type a, nor the aggregation for events of type c can complete, no further event processing

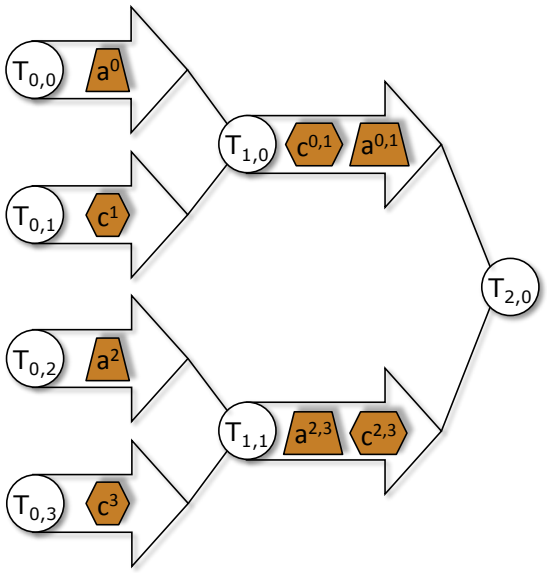


Figure 4.20: *An example of an event stream that uses superset channel identifiers, which can cause interlocking aggregations.*

is possible[3]. This results from the fact that the root place receives events with channel identifiers for leaf place index sets $\{0, 1\}$ and $\{2, 3\}$ respectively, while the events actually only include information of leaf index sets $\{0\}$, $\{1\}$, $\{2\}$, and $\{3\}$ respectively. The GTI prototype aborts aggregations if it detects that a queue tree allows no further event processing. In addition, it also aborts all aggregations after a configurable timeout. An extension of the GTI prototype supports additional encoding for channel identifiers to detect and consider patterns in superset identifiers such as odd/even leaf identifiers.

4.6 Previous Publications

Previous publications describe parts of GTI and its underlying abstraction. A first publication on GTI [78] briefly describes an early design of the infrastructure. This publication focuses on the communication system, in particular the MPI-based means of communication. The study presents basic performance measurements for synthetic communication loads. In particular, this early work excludes intralayer communication and broadcasts.

A second publication [72] describes the actual design and abstraction of GTI in natural language. It lacks the formal definitions for the abstraction terms, as well as the event-flow definitions that this document presents. Additionally, this document introduces the key algorithms of the weaver, as well as a main loop for tool places.

A first publication on order preserving event aggregation with channel identifiers [74] elaborates the basic queuing algorithm to which this document refers. Additionally, the previous publication introduces channel identifiers and queue trees. A key difference to this document is the description of the channel tree algorithms that were previously only described in natural language. The more detailed description with actual algorithms in this document also enables the detailed time complexity investigation of the previous section. Finally, the combination of a full main loop for tool places and the order preserving event aggregation algorithms enables a far more detailed description of the overall workings of tool places.

A further publication [75] highlights the rationale for intralayer communication along with a performance study of this functionality for MPI point-to-point matching. In particular, the former publication

[3] The aggregation on the first tool layer could be avoided since it replaces single events; in that case a binary TBON layout with 8 application processes still provides a similar scenario.

lacks the incorporation of intralayer communication into tool drivers and the adaption of the shutdown handling. Most importantly, this document introduces the event-flow definitions that apply to intralayer communication along with descriptions of how the weaver algorithms handle intralayer communication. At the same time, the previous publication provides deeper insight into communication modules for intralayer communication.

Finally, a last publication [126] details application crash-handling for GTI and the shared memory communication protocol that this design uses. This document summarizes GTI's application crash-handling approach, while the previous publication serves as a reference for comparisons and a detailed investigation of this technique.

4.7 Runtime Verification Requirements

The GTI prototype meets all of the MPI runtime verification requirements from Section 3.2 (page 22):

- Its instantiation system provides an instrumentation system that is tailored for the use case of the tool,
- A topology of additional tool places provides event filtering and aggregation capabilities,
- Exchangeable communication modules provide flexibility in both the means of communication and its timing, and
- An application crash-handling scheme detects a potential application crash and ensures that event processing can continue,

In addition, the proposed abstraction overcomes the drawbacks that a development with existing tool infrastructures would have entailed:

- Lack of an integration between instrumentation and tool functionality,
- Different interfaces for front-end, back-end, and tool hierarchy tool analyses,
- No event order between different event streams, and
- Unsatisfactory support for point-to-point event matching tasks.

The analysis-centric abstraction that this thesis proposes combines: An integration of a tool infrastructure provided instrumentation system with tool analyses, as well as a single interface for all tool analyses. Thus, tool developer do not need to manually develop/apply and couple an instrumentation system, but they rather only need to describe the interaction between instrumentation and tool analysis to the tool infrastructure. Additionally, in the proposed abstraction, all tool analyses describe their interfaces to the tool infrastructure. The infrastructure then triggers analyses whenever their input events arrive on a place. This abstraction ensures that tool analyses can be mapped freely onto places. This overcomes the fixed categorizations into front-end, back-end, and intermediate layer code that existing tool infrastructures apply. Such a situation increases flexibility in component reuse and tool integration.

The algorithms for order preserving event aggregation then overcome the limitations of the stream-based concepts that existing TBON tool infrastructures apply. In the proposed abstraction, a tool infrastructure can automatically preserve event order, irrespective of whether events are of the same type (same stream) or not. This feature simplifies the development of tools that must consider such order, since they can rely on the tool infrastructure, rather than implementing their own timestamping mechanisms. The intralayer communication that the proposed abstraction incorporates into its event-flow then enables efficient handling of tasks such as point-to-point matching.

In summary, the proposed abstraction formally introduces a novel concept for parallel tool infrastructures that overcomes the described drawbacks of existing approaches. This formal description provides the concept in a use case and programming paradigm independent manner, as to enable a high

degree of applicability. The prototype implementation GTI implements this abstraction—with various extensions—for use cases that target MPI applications. With that, GTI provides a tool infrastructure that supports the MPI runtime verification use case. Thus, a tool development effort can then exclusively focus on the actual tool functionality.

Current efforts extend the GTI prototype to provide CUDA instrumentation capabilities for runtime verification of CUDA applications. Additionally, a study [72] applies a GTI-based tool to gather profiling information for MPI communication, while a further GTI-based tool prototype enables an online performance workflow [89, 165].

5 A Distributed MPI Runtime Verification Concept

> *Scientific computing provides predictions that are increasingly important not just for research but also for decision-making on issues of great significance to society, including economic policy, environmental regulation, and the safety and performance of such things as cars, airplanes, and buildings. Yet the parallelism that makes much of this computation practical makes it difficult to build correct programs.* [142]

This chapter presents concepts for scalable runtime verification of MPI applications, which avoid a loss of precision, i.e., that avoid false positives and false negatives. The Marmot Umpire Scalable Tool (MUST) provides a prototype implementation for these concepts. It uses GTI as its tools infrastructure and carefully considers the advantages and disadvantages of previous MPI runtime verification tools; especially Umpire [159], Marmot [99], and TAC [120]. The implementation of MUST combines most of the existing checks of Marmot and Umpire and favors precise checks of Umpire over less precise checks of TAC. Correctness analyses that involve two or more processes are the focus of this chapter, which includes analyses such as type matching for point-to-point and collective operations, validations for consistent arguments in collective communications, and deadlock detection. Previous concepts for these analyses either lacked scalability (e.g., Marmot, Umpire, and ISP) or precision (e.g., hash based type matching in TAC and the NEC/SX library, as well as timeout based deadlock handling in TAC and Marmot). New concepts for these challenging checks advance the state of the art for runtime MPI verification tools and serve as a use case to test and verify the prototype implementation GTI.

Section 5.1 summarizes the overall architecture of the MUST prototype. Afterwards, Sections 5.2 and 5.3 present the concepts and the designs that handle MPI point-to-point and collective operations. The deadlock detection in MUST combines a graph-based deadlock detection, timeouts, and the use of a transition system to provide both detailed error reports and scalability. Section 5.4 presents this transition system and relates it to existing transition systems for MPI verification. A design for a distributed implementation of the transition system provides the scalability for this concept. Based on the transition system, Section 5.5 details an overall deadlock detection system that incorporates graph-based deadlock detection and timeouts. Section 5.6 then compares the contributions of this chapter to previous publications. A comparison of the proposed runtime verification concepts to existing MPI runtime verification approaches concludes this chapter (Section 5.7).

5.1 Tool Design

GTI provides mechanisms to instantiate a tool for a given use case, to instrument an application, to implement tool internal communication, and further types of common or specialized infrastructure functionality. As a result, the MUST prototype only needs to provide its actual functionality as a set of GTI modules and specifications. These specifications describe MPI as a set of hooks, tool internal hooks for MUST's internal communication, and analysis-hook mappings. This section first classifies modules of MUST's design into different functionality classes and uses this classification to define *module packages*. One of these module packages defines all correctness checks of MUST. For this *check package*, this section summarizes the individual correctness analyses that involve multiple processes (non-local) and defines their relation to other MUST modules. The remaining module packages define utility modules that serve for process identification, MPI resource tracking, logging, and event conditioning. Users can manually specify a tool layout, and place correctness modules onto the layers of the layout. However, to provide a scalable tool instance that exhibits low overhead, MUST defines standard configurations that simplify tool usage.

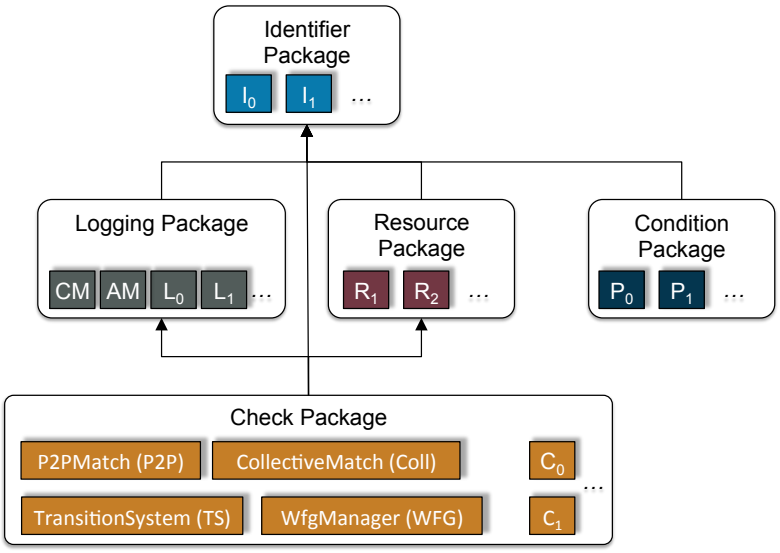


Figure 5.1: *MUST module packages (rounded boxes), their overall dependencies (arcs), and key modules (colored boxes).*

5.1.1 Module Packages

Figure 5.1 sketches module dependencies with arcs between module packages. The modules of the *identifier package* provide basic information on MPI ranks and call locations, e.g., a call name or a call stack. Thus, most modules require these services to identify which MPI rank created an event. As a result, most modules have a module dependency to one or multiple modules of the identifier package. Modules of the *check package* create *correctness messages* when they detect a failure. Dependencies to modules of the *logging package* provide a service to report these correctness messages to the tool user. In addition, some check modules can reuse information from other check modules, i.e., these modules can have internal dependencies. The *resource package* provides information on handles that identify MPI datatypes, communicators, groups, requests, and operations. The individual modules in this package track all default instances of each of these resources and monitor resource creation, destruction, and modification. Check modules use module dependencies to modules of the resource package to retrieve detailed information about involved MPI resources. Some modules of the resource package require information on a different type of resource, e.g., a module to track MPI communicators requires information on MPI groups. Thus, modules in the resource package may depend on each other. The *condition package* provides modules that evaluate whether MPI operations satisfy a specific condition. If so, they inject a specific event that another module uses (condition pattern from Section 4.3.1 on page 53). Condition modules should run on a lower-level hierarchy layer to maximize their effect. As a result, default layouts of MUST place modules of the condition package onto the application layer.

5.1.2 Utility

When an event triggers an analysis of a MUST module, the module usually requires information on the MPI rank that created the event. As an example, a module that checks whether a communication operation uses a correct MPI datatype requires rank information to compare the given datatype to valid datatypes for this rank. In addition, if a check detects an MPI usage error, it should provide the tool user with as much detail on the defect as possible. One such piece of information is the name of the call that caused the usage error and preferably additional information such as a call stack or iteration counters. The modules and operations of the identifier package provide such information. MUST associates a parallel identifier (pId) with MPI events to provide MPI rank information to all analyses that request this information. The implementation stores pIds with 64 bit values. This capacity suffices to directly store

an MPI rank and provides additional storage capacity for information on further parallel paradigms, e.g., a thread number. An additional location identifier (lId) provides analyses information on the call site that created an MPI event. Again the implementation uses 64 bit values for lIds. However, such a value cannot store arbitrary textual strings or even a call stack. Thus, MUST uses mappings to associate call location information with $(\mathrm{pId}, \mathrm{lId})$ pairs. Appendix B.1 (page 167) details the designs to create these identifiers and to forward call site information to other layers of the tool layout.

Modules of the check package require a system to report detected failures, suspicious behavior, or to provide other forms of information to the tool user. The modules of the logging package provide services to create correctness messages and to store them for investigation. A module dependency to the *CreateMessage* module (CM in Figure 5.1) provides access to an interface that introduces new correctness messages. The *CreateMessage* module uses event injection to introduce events that represent correctness messages. Logging modules ($L_0, L_1, \ldots$ in Figure 5.1) are mapped to the hooks that inject these events, such that these logging modules can receive and handle messages. GTI forwards events towards any layer with a logging module, the *AggregateMessage* module (AM in Figure 5.1) combines similar messages to condense the events that arrive at a logging module. Thus, the MUST prototype can provide users condensed messages—e.g., ranks 0–1024 violated a specific MPI restriction—in various output formats. Default instances of MUST use an HTML format for this purpose. Appendix B.2 (page 168) details these modules and the hook that introduces events for correctness messages.

MPI provides resources such as communicators, requests, datatypes, process groups, and reduction operations to application developers. Handles identify these objects in MPI operations, while the MPI implementation's internal representation of these objects is not accessible to an application. Many correctness checks directly apply to these objects—*resources* in MUST terminology—to verify their correct creation, destruction, or state. As an example: To ensure that a target rank in an `MPI_Send` operation is a valid rank within the given communicator. Additionally, several correctness checks investigate these resources as part of their analysis, e.g., point-to-point matching evaluates the communicator resource to translate ranks. MPI provides limited query functions to retrieve state information for a resource handle, e.g., it provides no functions to query whether a handle is valid. Further, queries to MPI are only available on the MPI processes directly, but not on other layers of a tool layout. Thus, the modules of MUST's resource package provide information on MPI resources instead. Each resource package module tracks the state of all resources of a single type, e.g., all MPI datatypes. The tracking captures predefined resources, e.g., `MPI_INT`, and user-defined resources, e.g., a derived datatype constructed with `MPI_Type_struct`. Check modules retrieve access to the interface of a resource package module with a dependency. As a consequence, modules of the resource package run on the same layers as the modules that require their information. Resource package modules map their analyses to all MPI operations that create, destroy, or modify a resource. These mappings ensure that all instances of a resource package module provide correct and up-to-date information.

Datatype, request, communicator, and group tracking modules provide services to forward resource information along the intralayer communication direction. Type-matching checks for point-to-point communication operations require such services since they forward information on MPI operations along the intralayer direction (rather than the primary communication direction). To support this use case, the resource package modules provide services that replicate the state of a resource onto another place of the same layer. This scheme does not update the replicas when successive state changes occur, as to avoid extensive communication overheads. Later sections detail that modules that require this intralayer replication will not use stateful information of such resources.

The modules of the condition package serve as event filters. Analysis-hook mappings of modules that do not run on the application layer require tool communication. This communication unnecessarily increases the tool overhead, if a module is only interested in a subset of these events, i.e., events that satisfy a given condition. One such example is the module that matches point-to-point operations for type-matching and deadlock detection. This module requires information on completed nonblocking receives that use `MPI_ANY_SOURCE` as their source argument (wildcard receives). A direct mapping of a module to all MPI completion operations that provide this information, e.g., `MPI_Wait`, would cause

unnecessary events forwarding. Thus, the modules of the condition package implement such conditions and inject events whenever a condition is met. This technique efficiently reduces tool communication. Appendix B.3 (page 169) details how a condition package module filters events of MPI completion operations to extract conditional information.

5.1.3 Correctness Checks

The implementation of MUST provides a variety of—local—correctness checks that require information from a single MPI process only. Figure 5.1 summarizes the modules that implement these checks as $C_0, C_1, \ldots$ in the check package. It includes most of the checks that Marmot [97] provides and includes communication buffer overlap checks [125] that consider basic and derived MPI datatypes. While these types of checks can impose noticeable overheads, their performance usually does not depend on application scale (scale can impact an applications problem size and thus indirectly the overheads of these checks).

This document focuses on—non-local—correctness checks that use information from multiple MPI processes, since their implementation strongly impacts tool scalability. As a result, existing tools either provide scalable non-local checks that can exhibit false positives/negatives to simplify the design of these checks, e.g. TAC; or have limited scalability without a loss in precision, e.g., Umpire. The non-local checks that this chapter introduces aim at a combination of both precision and scalability. Figure 5.1 summarizes the MUST modules that implement these non-local checks:

P2PMatch: Matches point-to-point messages, detects lost messages, and issues type matching checks for MPI datatypes (P2P in Figure 5.2);

CollectiveMatch: Matches events of collective operations, checks for consistent arguments, and issues type matching checks for MPI datatypes (Coll in Figure 5.2);

TransitionSystem: Analyzes conditions under which MPI operations return the control flow as a transition system (TS in Figure 5.2); and

WfgManager: Applies a Wait-For Graph (WFG) supported deadlock detection that uses the state of the *TransitionSystem* module as input (WFG in Figure 5.2).

The *P2PMatch* module analyzes point-to-point operations both to detect lost messages, as in the example of Figure 2.8 (page 10), and to detect type mismatches, as in the example of Figure 2.1 (page 8). The module uses intralayer communication to exchange information for message matching and type matching. Thus, standard MUST layouts place this module on the first non-application layer, since this layer provides the highest degree of distribution. Section 5.2 details this module and its analyses.

The *CollectiveMatch* module is an aggregation module that implements checks for collective operations in the tool hierarchy. As a result, this module runs on all hierarchy layers of the tool for standard layouts of MUST. Collectives such as `MPI_Alltoallv` require type matching checks that limit the efficiency of the aggregation module. Thus, the aggregation module can exchange type matching information for such operations with intralayer communication too. Section 5.3 details this module and its hierarchical checks.

Finally, the *TransitionSystem* module implements a transition system that emulates the wait-for semantics of all MPI operations in their execution order. If an application deadlocks, the transition system will also arrive in terminal state that indicates deadlock. The module uses information on point-to-point and collective operations for its analysis and runs on the same layer as the *P2PMatch* module. The *TransitionSystem* module exchanges state information with intralayer communication and with the primary communication direction to synchronize state transitions. Utility modules then aggregate and manage these synchronization events. Section 5.4 presents the theoretical background for the *TransitionSystem* module along with a concept and design for a distributed implementation. The *WfgManager* module then applies graph-based deadlock detection to a current state of the *TransitionSystem* module. It triggers deadlock detection in fixed intervals, assures a consistent state of the *TransisitionSystem* module

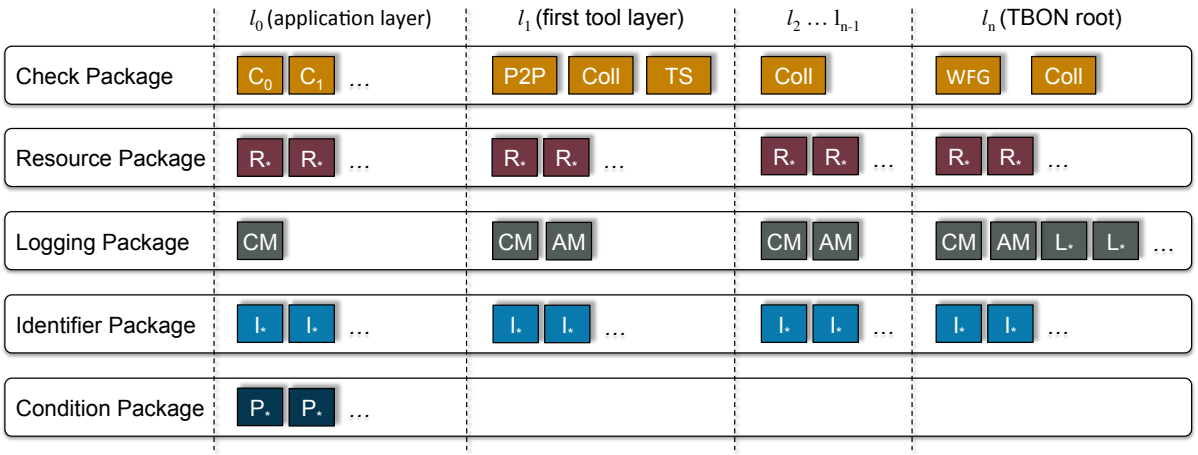


Figure 5.2: *An illustration of important layers for common MUST instantiations, along with the modules that they use.*

instances, applies a deadlock search, and if a deadlock was found, reports it to the user. The module is centralized, i.e., must be placed on a layer with exactly one place. Section 5.5 details this graph-based deadlock detection, its modules, and its tool internal communication.

5.1.4 Instantiation

MUST provides utilities that create standard layouts for default use cases. This includes the use of centralized non-local checks, which serve as a reference implementation and that improve upon the non-local checks of Umpire. Such a layout uses a single extra layer with a single place to execute all non-local correctness checks.

Figure 5.2 summarizes the module placement of MUST layouts that use the distributed non-local checks that this thesis proposes. These layouts use a TBON topology, i.e., a layer tree that is a list and whose final layer uses exactly one place. Additionally, the MUST utilities provide control over the fan-in that the resulting TBON topology uses. The application layer executes all local correctness checks ($C_0, C_1, \ldots$ on l_0). In addition, the application layer executes the modules of the condition package to maximize their filtering capabilities. MUST's utilities map the *P2PMatch* (P2P) and *TransitionSystem* (TS) modules onto the first non-application layer, as to distribute these checks across as many places as possible. The TBON root then executes one or multiple logging modules ($L_*, L_*, \ldots$), the *WfgManager* (WFG) module, and a module that enables the use of the *CollectiveMatch* (Coll) aggregation module on all non-application layers. Module dependencies then automatically place the resource package modules, the identifier package modules, and the *CreateMessage* module (CM) onto all layers that require them.

5.2 Point-to-Point Analysis

Non-local correctness checks that apply to MPI point-to-point operations are type matching (e.g., the example of Figure 2.1 on page 8) and the detection of lost messages (e.g., the example of Figure 2.8 on page 10). Both checks require information on pairs of matching point-to-point operations. The *P2PMatch* module of MUST serves for this purpose and monitors all point-to-point operations in order to apply the MPI message matching rules to them. The module triggers type matching checks when it determines that a pair of operations matches and it reports any lost messages when it observes that an application ends its MPI usage. With point-to-point communication, pairs of processes exchange messages, while the communication pattern of an application can be dynamic, i.e., a-priori knowledge on communicating process pairs may not be available. Thus, as Section 4.3.4 (page 58) illustrates, a TBON-based implementation of point-to-point matching [72] can exhibit higher overheads on higher-level tool lay-

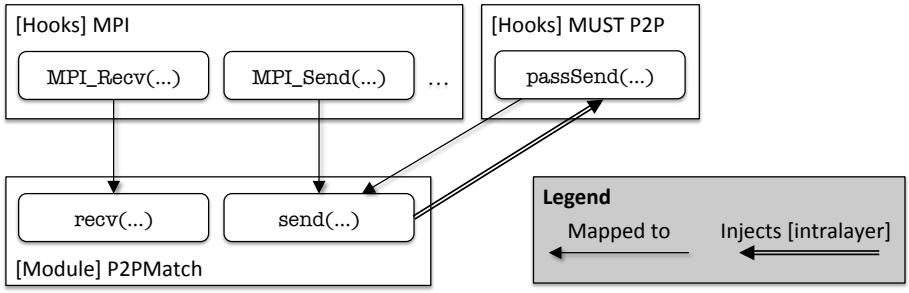


Figure 5.3: *A* module-hook *chart that illustrates how the proposed tool infrastructure abstraction enables the implementation of a* P2PMatch *module for distributed analysis of MPI point-to-point operations.*

ers. Thus, the *P2PMatch* module uses intralayer communication to avoid scalability limitations. MUST places this module on one layer of the tool layout and exchanges point-to-point matching information between module instances within this layer (Figure 5.2).

Intralayer communication based point-to-point message matching overcomes the scalability limitations of previous runtime verification tools such as Umpire [159] and ISP [156]. At the same time, this technique resembles the post-mortem matching approaches of tools such as Scalasca [53] and Vampir [21]. The implementations of these tools communicate between pairs of places—MPI application processes for Scalasca or an MPI-based analysis application for Vampir—to distribute MPI point-to-point matching information. However, both of these implementations do not provide TBON services neither do they apply correctness checks to pairs of matched point-to-point operations. Section 5.4 then details how point-to-point matching in combination with synchronization messages within a TBON topology allows complex analyses such as the execution of a transition system for deadlock detection.

5.2.1 Module Design

Figure 5.3 illustrates the analyses of the *P2PMatch* module and the hooks to which it is mapped. This chapter uses *module-hook* charts as in Figure 5.3 to detail how MUST uses GTI; to represent modules and their key analyses; to introduce hooks for MPI operations or tool internal communication; to visualize analysis-hook mappings; and to highlight which hooks an analysis uses to inject events. Boxes with a "[Hooks]" prefix in their label (upper half of the figure) summarize a set of related hooks, e.g., hooks for MPI operations. The rounded boxes within them represent individual hooks—and potentially their arguments. Boxes with a "[Module]" prefix in their label (lower half of the figure) represent modules and their analyses as embedded boxes with rounded edges. Solid arcs from a hook to an analyses highlight that the analysis at the head of the line is mapped to the hook at the tail of the line. Particularly, the direction of the arcs illustrates the information flow. Dashed arcs from an analysis to a hook illustrate that the analysis injects events along the primary communication direction with this hook. Similarly arcs with two parallel, solid lines illustrate event injection along the intralayer communication direction and arcs with two parallel, dashed lines illustrate event injection along the broadcast communication direction.

The *P2PMatch* module provides the `send` and `recv` analyses to observe send and receive operations of MPI. Figure 5.3 illustrates that these analyses use mappings to the hooks for MPI operations such as `MPI_Send` and `MPI_Recv` respectively. The module does not require a specific handling for distinct send modes (buffered, synchronous, or ready), since it only matches pairs of point-to-point operations. Thus, MUST can map the `send` analysis to the hooks for these operations as well. The same holds for nonblocking point-to-point operations such as `MPI_Isend`. The implementation of MUST stores information on the request handle that is associated with nonblocking point-to-point operations for its correctness reports and to interface the *P2PMatch* module with the *TransitionSystem* module in subsequent sections. Finally, for point-to-point matching purposes, a tool can handle persistent point-

to-point operations like nonblocking operations, i.e., as a send or receive operation with an associated request handle. For simplicity, Figure 5.3 excludes the additional analyses that monitor invocations of `MPI_Start` and `MPI_Startall`, which trigger these persistent point-to-point communications.

This chapter uses the term *match layer* to refer to the single layer that executes the *P2PMatch* module, which is the first non-application layer in standard layouts of MUST. For p application processes, let $P = \{0, 1, 2, \ldots, p-1\}$ be the set of process identifiers (ranks). For a match layer with k places, the module instance on the i-th place of the layer receives information on all point-to-point operations for a subset $K_i \subseteq P$ (Such that $\bigcup_{i=0}^{i<k} K_i = P$ and $\forall i, j \in \{0, 1, 2, \ldots, k-1\}, i \neq j$ holds $K_i \cap K_j = \emptyset$). The analysis-hook mappings in Figure 5.3 allow GTI to forward information on all point-to-point operations to the instances of the *P2PMatch* module. The *P2PMatch* module applies a domain decomposition to distribute the matching workload, such that the i-th place of the match layer matches all operation pairs that include a send operation transferring data to a process in K_i. This decomposition covers all operation pairs and assigns each pair a unique place of the match layer. Thus, if an instance of the *P2PMatch* module observes a send operation with a destination rank that is in K_i, then it forwards information on the operation to the i-th place of the match layer. Figure 5.3 illustrates the `passSend` hook that the *P2PMatch* module uses to forward this information with intralayer communication. In summary, a *P2PMatch* module instance on the i-th place of the match layer perceives information on receive operations along the primary communication direction with the analysis-hook mappings of the `recv` analysis in Figure 5.3. Information on matching send operations then either arrives along the primary communication direction (if the send also originates from a process in K_i) or along the intralayer communication direction. The point-to-point operation matching rules of MPI consider event order. For each pair of processes, send and receive operations must be matched in the order in which the two application processes issued them. The communication of GTI does not violate this order, since both the primary and the intralayer communication direction preserve the order of the events that they communicate.

The proposed domain decomposition for the *P2PMatch* module forwards send operations with intralayer communication, since MPI's push semantic guarantees that the target process of all send operations is specified. The source rank for wildcard receive operations is unknown until the receive completes. Forwarding receive operations—instead of send operations—would require additional considerations.

5.2.2 Data structure

The *P2PMatch* module must store information on unmatched send and receive operations and uses a data structure for this purpose. The data structure only holds the operations that the module instance will match, i.e., it excludes operations that the module forwards with intralayer communication. The message matching rules of MPI specify that a send and a receive operation match if they both use the same communicator, their source and target ranks are compatible, and if they both use the same message *tag* (or the receive uses `MPI_ANY_TAG`). Finally, MPI considers message order to resolve situations in which multiple send or receive operations could match.

For each rank that issues point-to-point operations and for each communicator, the *P2PMatch* module uses a separate *matching table* to store unmatched operations. Each matching table uses one list of send operations for each target rank of a send and one list of receive operations for each source rank of a receive. This allows the module to look up a potentially matching send/receive operation for a new receive/send operation as follows: With $\mathcal{O}(\log k)$ time it can find the matching table for a total of k communicators; with an additional $\mathcal{O}(1)$ it can look up the respective target or source operation list; and with a cost of $\mathcal{O}(l)$ it can evaluate all l operations in the list. However, $\mathcal{O}(1)$ time to look up the matching tables for an issuer rank or the operation list for a source/target rank requires at least $\mathcal{O}(p)$ memory to allocate all entries. Thus, the MUST implementation uses a map data structure to associate matching tables with issuing ranks and an additional map to associate a source/target rank with an operation list. These maps add an additional $\mathcal{O}(\log p)$ search time if an application communicates with all other processes. If processes communicate with $n \ll p$ processes only, then this search time reduces

to $\mathcal{O}(\log n)$. The MUST implementation uses no specific operation lists per tag, to maintain event order for `MPI_ANY_TAG` support. Future implementations could optimize this handling, especially if an application does not use `MPI_ANY_TAG`.

In order to correctly handle wildcard receives, each matching table uses an additional *wildcard receive list* that stores these operations, since their source rank is not known. The use of this extra list requires additional algorithmic consideration to preserve the order of receive operations. Particularly, if a wildcard operation is present in a matching table, MUST's implementation will add any subsequent receive operations of the same communicator temporarily to the wildcard receive list too. At a later time, when the matching of the wildcard receive is determined (below), the implementation will re-queue all non-wildcard receives in the wildcard receive list to their respective source-rank-specific lists. This handling ensures that the matching can correctly consider the order between wildcard and non-wildcard receives. In the presence of wildcard receives, MPI implementations must also preserve the order of receive events. This handling can also impact their overheads for such a scenario, but is dependent on the particular implementation.

If a *P2PMatch* module instance receives information on a new send or receive operation that matches an operation in one of its matching tables, the module removes this entry from the table. Otherwise, if no matching operation is available, it adds the new operation to the table. However, for wildcard receives, multiple *potentially* matching send operations may exist. In that case, the MPI implementation decides which send matches. The *P2PMatch* module must use the same matching decision, as to provide a correct analysis [69]. Therefore, the module uses information from completed receive operations to retrieve this information from the MPI implementation. Appendix B.3 (page 169) details how MUST retrieves this information. When the *P2PMatch* module finds a send that could potentially match a wildcard receive, it postpones the matching until it receives the matching decision from the MPI implementation. This handling can delay the processing of successive operations. A rare situation arises if an application never issues a completion operation for a nonblocking wildcard receive operation, or deadlocks within the completion operation that returns this information. In that case, if at least one matching send for the operation exists, MUST fails to retrieve the update on MPI's matching decision. The centralized reference implementation of MUST implements a probing/deciding technique [76] to handle such scenarios, whereas the distributed modules of MUST do not provide such an extension yet.

5.2.3 Checks

Point-to-point matching provides input for the transition system in subsequent sections. In addition, whenever the *P2PMatch* module matches a send and a receive operation, it can check whether the type signatures of the two operations match. The MPI specification requires that the type signature of the send operation is a prefix of the type signature of the receive operation. Type matching algorithms in MUST [125] require $\mathcal{O}(1)$ matching time if both type signatures use a single basic MPI datatype only. The matching time for more complex types then depends on their respective structure.

The MPI standard requires that no unmatched point-to-point operations remain when all application processes issued `MPI_Finalize`. The MUST implementation detects such operations as unmatched entries in the matching tables that exist after all instances of the *P2PMatch* observed `MPI_Finalize`, while no outstanding intralayer communication exists. Lost message situations as in the example of Figure 2.8 (page 10) highlight that the presence of unfinished MPI requests is no requirement for the presence of an unmatched message. As a result, checks that detect the presence of unmatched messages provide valuable information to application developers. Especially, since some MPI implementations may silently tolerate such a defect, while others can hang or fail when the application issues `MPI_Finalize`. The *P2PMatch* module lists all unmatched point-to-point operations by traversing its matching tables and provides information on the invocations of the respective operations to aid developers in the removal of this defect.

MPI Operation Type	GTI Communication and Handling	*P2PMatch* Cost
Any send	$\mathcal{O}(1)$	$\mathcal{O}(\log p + \log k + l)$
Any non-wildcard receive	$\mathcal{O}(1)$	$\mathcal{O}(\log p + \log k + l)$
Any wildcard receive	$\mathcal{O}(1)$	$\mathcal{O}(\log p + \log k + p \cdot l)$
Completion operation with q requests	$\mathcal{O}(q)$	$\mathcal{O}\big(q \cdot (\log p + \log k + l)\big)$

Table 5.1: *Time complexities for the analyses of the* P2PMatch *module, along with the respective handling and communication complexities to provide their input (p application processes, k MPI communicators, l list size of a matching table slot).*

5.2.4 Time Complexities

Table 5.1 summarizes worst case time complexities for the analyses of the *P2PMatch* module, along with worst case time complexities for the handling in GTI that provides the input to these analyses. The "GTI Communication and Handling" column lists the time complexities for GTI's instrumentation, communication, and handling. GTI's instrumentation requires $\mathcal{O}(1)$ to intercept any point-to-point operation on the application, serialize the fixed number of arguments that the *P2PMatch* module requires into a buffer for communication, and to communicate the buffer to the match layer (assuming that the target place of the match layer is idle and that bandwidth and latency are independent of the tool layout). The receiving place requires $\mathcal{O}(1)$ to receive the buffer, unpack it, and to trigger the respective target analysis of the *P2PMatch* module (assuming no active aggregations). The intralayer communication that forwards information on send operations also requires $\mathcal{O}(1)$ time per operation (target place is idle, tool layout independent bandwidths and latencies).

The "*P2PMatch* Cost" column of Table 5.1 summarizes the time complexities for the analyses of the *P2PMatch* module. For non-wildcard communication operations, these analyses either forward an operation with $\mathcal{O}(1)$ time to another module instance on the same layer, or handle the operation with a cost of $\mathcal{O}(\log p + \log k + l)$ for p processes, k communicators, and a list length of l (as described previously). The size of l depends on the point-to-point operation series that an application uses and the order in which place drivers receive events. The *P2PMatch* module only needs to iterate over an operation list until it finds a match. Thus, the performance impact of long operation lists can be low. If an application uses a single tag only, l may still not be constant. However, a search in a non-empty list will always produce a match for the first list entry. Thus, with a single tag, the analysis cost in the *P2PMatch* module becomes $\mathcal{O}(\log p + \log k)$. If furthermore, each application process only communicates with n processes, while the application only communicates in the default communicators, then the time complexity becomes $\mathcal{O}(\log n)$. The *P2PMatch* module must search through all send operation lists to determine whether a potentially matching operation for a wildcard receive exists. Thus, for p application processes, the module requires $\mathcal{O}(\log p + \log k + p \cdot l)$ time to search through up to p lists with an average list length of l. The amount of non-empty lists of send operations depends on the number of processes that send to the issuer of the wildcard receive. This count can be low for applications that define their communication patterns on stencil computations, i.e., that use neighborhood communications.

Completion operations such as `MPI_Waitall` use an array of q requests. The *P2PMatch* module receives information on all successful completions of MPI requests that are associated with nonblocking wildcard receives. For each such receive, the module updates the source rank of the receive operation and tries to find a matching send operation, which must be issued by the application, but may not yet have arrived at the *P2PMatch* module instance. This handling requires $\mathcal{O}\big(q \cdot (\log p + \log k + l)\big)$ time if all requests are associated with wildcard receives and for an average operation list length l.

The time that an MPI implementation requires to implement a point-to-point operation is $\mathcal{O}(1)$ in the best case. In summary, the time complexities of the *P2PMatch* module allow operation series and processing orders that could require higher time complexities than the original MPI operations. Particularly, some of these factors—list length l and the $\log p$ mapping time—can increase with scale. Thus, the

MUST implementation could incur noticeable overheads that increase with scale. Chapter 6 evaluates whether such scenarios appear in actual MPI applications, as well for synthetic kernels. However, depending on the internal data structures of the MPI implementation, outstanding point-to-point operations can impose increased handling costs too, since an MPI implementation must consider operation order in the presence of multiple tags, wildcard tags, and wildcard receives too. Additionally, GTI communicates at most two events of a fixed (small) size for each point-to-point operation (for sends one event on the primary and one event on the intralayer direction), whereas an MPI implementation must communicate a message buffer of a given payload for each pair of operations, which will usually exceed GTI's payload. Finally, MPI implementations can require additional costs to guarantee progress of nonblocking point-to-point operation. Thus, additional terms such as the number of open nonblocking point-to-point messages influence MPI implementation costs to handle point-to-point operations in practice [68].

5.3 Collective Analysis

Collective communication operations of MPI provide data transfers and reduction operations between groups of processes. MPI communicators specify these process groups. For a collective communication to take place, all involved processes issue a collective operation of the same name with a communicator argument that exactly includes the involved processes. The term *collective* refers to the overall communication between all members of a process group and the term *collective operation* refers to a single MPI call invocation of a member of the group. MPI usage errors for collective operations involve deadlocks, MPI datatype matching defects, and further kinds of inconsistent arguments. The *CollectiveMatch* module of MUST implements correctness analyses for these usage errors and employs the distributed detection scheme that this section presents. The module issues checks to detect inconsistent arguments and type matching defects. Additionally, it provides input for deadlock detection (subsequent sections).

Collective checks in related work either use a centralized place that matches collective operations of all processes, e.g., Umpire [159] and ISP [156], or they use additional collectives to check arguments of collective operations, e.g., TAC [120], MPI/SX [153], and MPICH-Coll [40]. While the former limits scalability, the latter influences synchronization properties of the MPI application, provides limited type matching checks, and was only combined with timeout-based deadlock detection approaches yet, i.e., has limited precision.

This section presents a concept for collective checks that operate in a TBON layout with a hierarchical scheme. The concept uses the observation that most checks for collectives are transitive, i.e., that it suffices to select one collective operation as a representative of a collective and to compare all other operations to this representative only. This section first details this hierarchical approach and highlights its limits for type matching checks that can specify non-transitive restrictions. Afterwards, a description of the *CollectiveMatch* module highlights how MUST implements hierarchical collective checks and how it employs GTI. Finally, this section details time complexities of the GTI handling and for the analyses of the *CollectiveMatch* module. This investigation requires a description of the data structures of the *CollectiveMatch* module, as well as a description of the individual correctness checks.

5.3.1 Representative Operations

Figure 2.3 (page 9) illustrates an example defect that involves inconsistent collective operation arguments. In the example, processes 0 and 1 specify different values for the root argument. An implementation for a correctness analysis to detect this failure must compare arguments of collective operations that belong to the same collective. A comparison such as "Do all processes use the same root argument?" is transitive. Thus, it is not necessary to compare all pairs of operations with each other. Rather, if collective operations $o_0, o_1, \ldots, o_n$ form a collective, then a set of operation comparisons between pairs of operations $C = \{(o_{i_0}, o_{j_0}), (o_{i_1}, o_{j_1}), \ldots, (o_{i_m}, o_{j_m})\}$ $(\forall k \in \{0, 1, \ldots, m\} : 0 \leq i_k, j_k \leq n)$ is guaranteed to reveal a potential inconsistency if: The transitive closure I^* of the set of compared operation indices $I \stackrel{def}{=} \{(i, j) : (o_i, o_j) \in C \text{ or } (o_j, o_i) \in C\}$ includes index pairs for all pairs of distinct operations

($I^* \supseteq \{(i,j) : 0 \leq i,j \leq n, i \neq j\}$). For transitive correctness properties, any comparison scheme that satisfies this condition is sufficient to detect the presence of a violation. As an example, the approach in Umpire uses one operation o_x of each collective as a representative and compares all other operations against o_x.

To provide a scalable and distributed comparison scheme, this thesis proposes a step-wise hierarchical comparison that operates on a tool layout with a layer l that uses exactly one place. Layer l and all its predecessor layers in the layer tree execute a hierarchical comparison as follows: If a place on these layers can receive operations $o_0, o_1, \ldots, o_n$ of a collective along the primary communication direction, it selects a representative o_x ($0 \leq x \leq n$) and compares all remaining operations that it receives to o_x. The place then forwards an event for operation o_x and removes the events for the other operations from the primary communication direction. Further, the place stores in the event for operation o_x that this event is a representative for $o_0, o_1, \ldots, o_n$. Places use this information to determine when they perceived all operations for a collective. If a place detects an inconsistency during a comparison, it reports it to the user[1].

MUST implements this scheme with the *CollectiveMatch* aggregation module that replaces events for collective operations with a representative event. A module on a layer l, using one place, enables the aggregation module on l and all of its predecessor layers. GTI then automatically applies the aggregation module during its module placement. Further, the MUST implementation modifies the scheme such that it only applies the aggregation if all input events are consistent. This allows the implementation to report multiple pairs of operations that violate an MPI restriction, which can simplify root-cause analysis.

The hierarchical comparison scheme satisfies the above condition that the transitive closure of all compared operations includes all pairs of distinct operations (without proof). As opposed to a centralized scheme on a single place, the hierarchical scheme provides an efficient distribution. For a k-ary TBON layout, and collectives that involve all processes, each place receives information on k operations and forwards information on one operation. Particularly, the workload on each place is independent of the application scale if k remains constant when scale increases.

The following checks apply to collective communications:

Call: Consistent type of operation, e.g., all operations are a `MPI_Bcast`, within one collective;

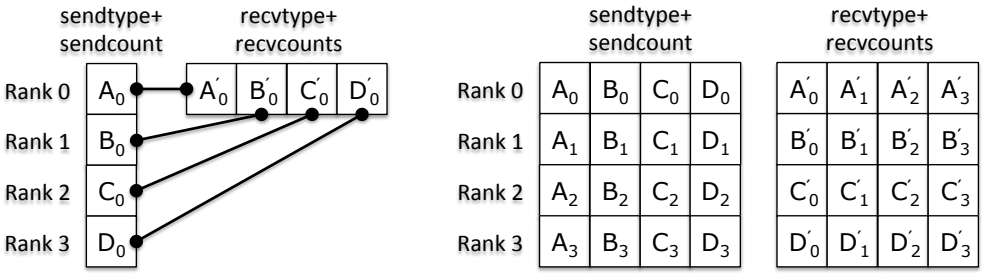
Root: Same root for all operations of a collective that uses a root process;

Op: Compatible reduction operator (`MPI_Op`) for collectives that use such an operator; and

Types: Type matching for all data transfers of the collective.

All of these checks are transitive with the exception of the type matching rules of the irregular collectives `MPI_Gatherv`, `MPI_Scatterv`, `MPI_Alltoallv`, and `MPI_Alltoallw`. Collectives such as `MPI_Gather` specify transitive type matching rules where the root of the collective and all other processes must use an identical type signature. Type matching checks for such collectives can use a transitive scheme. The irregular version of this collective (`MPI_Gatherv`), however, can specify distinct type signatures between the root process of the collective and each other participating process. Thus, for irregular collectives, a transitive type matching scheme is inapplicable. The *CollectiveMatch* module handles all transitive checks with the hierarchical scheme. For the collectives that use non-transitive type matching rules, the module uses intralayer communication to efficiently distribute type matching information (subsequent section). The above checks require $\mathcal{O}(1)$ time except for type matching, where the structure of the datatype influences the comparison cost. For simplicity, this section assumes $\mathcal{O}(1)$ time complexity for type matching, which allows comparisons for any predefined datatype, as well as for user-defined datatypes that use a single basic MPI datatype only. Thus, a comparison of a collective operation to a representative requires $\mathcal{O}(1)$ time.

[1] The implementation in MUST restricts checks such that higher hierarchy layers will not re-execute checks that lower hierarchy layers already executed, thus, correctness reports include no redundant violation reports.



(a) Equal type signature labels (e.g., A_0 and A'_0) and connected lines illustrate the pairs of type signatures that a runtime verification tool needs to compare for `MPI_Gatherv`.

(b) Pairs of type signatures to compare for `MPI_Alltoallv`. The illustration avoids the use of connecting lines between the 16 pairs of signatures for presentation purposes.

Figure 5.4: *Examples for non-transitive type matching rules for four application processes.*

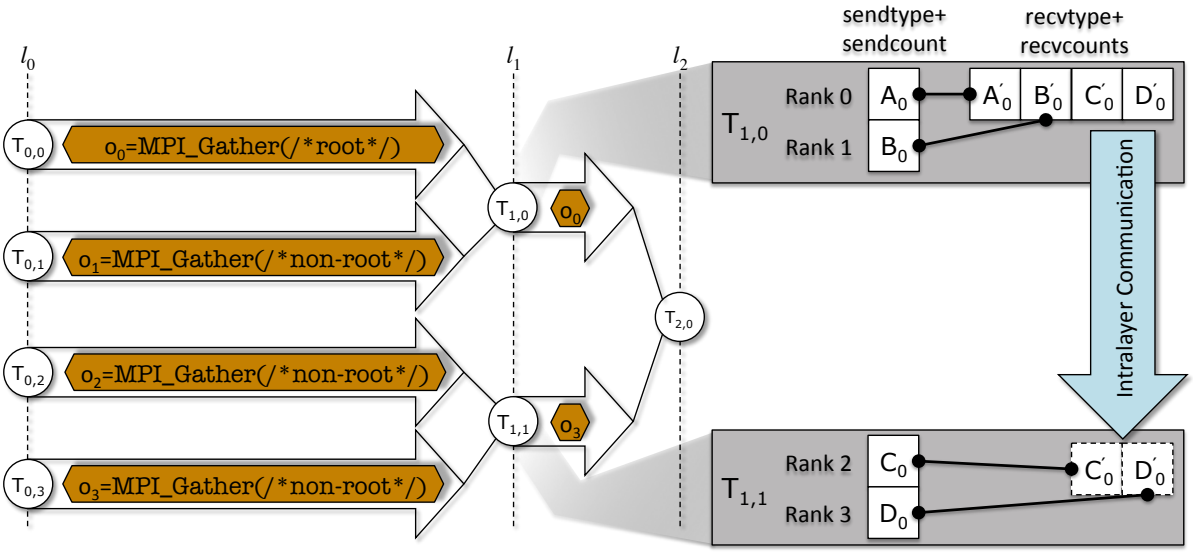
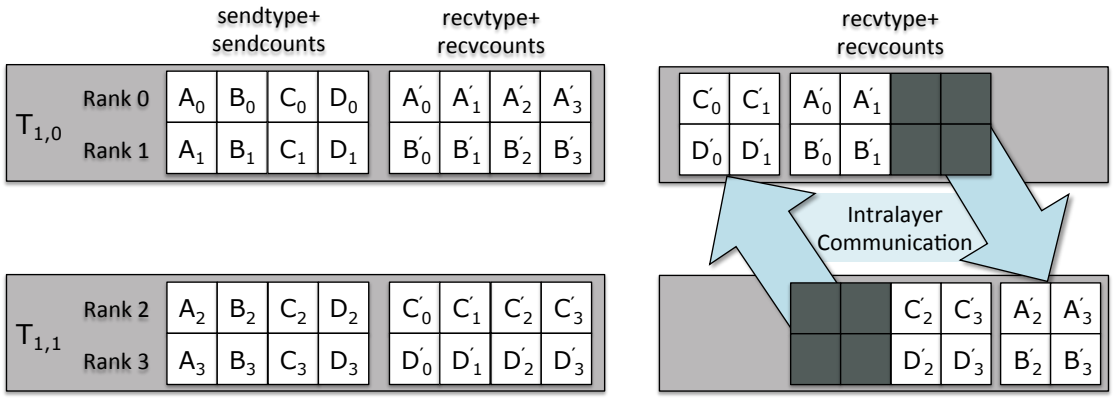


Figure 5.5: *An illustration of the data distribution for type matching data on two places $T_{1,0}$ and $T_{1,1}$, with events for a `MPI_Gatherv` collective. The distribution motivates the use of intralayer communication to forward type matching information (C'_0 and D'_0) from $T_{1,0}$ to $T_{1,1}$.*

Finally, the irregular collective operations `MPI_Allgatherv` and `MPI_Reduce_scatter` specify redundant information that allows transitive comparison. The MPI standard requires that all processes specify similar count arrays for `MPI_Reduce_scatter` and similar type signatures for the data gathering in `MPI_Allgatherv`. If k processes participate in a collective, then each process uses arrays with k entries for this redundant information. As a result, for collectives that involve all processes, correctness checks that ensure that the redundant data is consistent require $\mathcal{O}(p)$ time during each operation comparison.

5.3.2 Intralayer Type Matching

For four application processes, Figure 5.4 illustrates the type signatures for the send and receive transfers of the `MPI_Gatherv` and `MPI_Alltoallv` collectives. For `MPI_Gatherv` (Figure 5.4(a)), each process uses a specific type signature A_0–D_0 to send data to the root process, which is process 0 in the example. The root process specifies all receive type signatures A'_0–D'_0 in turn. Correctness analyses for MPI datatype matching rules must then compare A_0 with A'_0, B_0 with B'_0, and so forth. Figure 5.4(a)



(a) Distribution of type matching information across places $T_{1,0}$ and $T_{1,1}$ highlights that both places lack information for type signature comparison, e.g., C'_0 is not available on $T_{1,0}$.

(b) All-to-all intralayer communication to exchange relevant type matching information between $T_{1,0}$ and $T_{1,1}$.

Figure 5.6: *The distribution of type matching information for* `MPI_Alltoallv` *on* CollectiveMatch *module instances of the match layer (for the tool layout in Figure 5.5) motivates an all-to-all exchange with intralayer communication.*

highlights these signature pairs with lines. For `MPI_Alltoallv` all pairs of processes use distinct type signatures for their transfer. Figure 5.4(b) illustrates such a collective with four processes. The `MPI_Alltoallv` and `MPI_Alltoallw` collectives require p^2 type signature comparisons for p processes.

For a binary tree layout, Figure 5.5 illustrates the hierarchical correctness analysis of an `MPI_Gatherv` collective with the *CollectiveMatch* module. The left side of the figure uses the event queue representation from Section 3.4 (page 28) and illustrates collective operations as events o_0–o_3. The first place of layer l_1 ($T_{1,0}$) receives the root operation o_0 and a non-root operation o_1 for the collective. Thus, $T_{1,0}$ receives information on two send type signatures and all receive type signatures, as the right side of the figure illustrates. Place $T_{1,1}$, however, receives information on two send type signatures of o_2 and o_3 only (lower right of the figure). As a result, $T_{1,1}$ can apply no type matching checks for this collective. The *CollectiveMatch* module uses GTI's intralayer communication system to distribute type matching information within the match layer for such situations. For `MPI_Gatherv`, the place that receives information on the root of the collective scatters information on the receive type signatures to all other places of the layer. Figure 5.5 illustrates this for places $T_{1,0}$ and $T_{1,1}$ where the former sends the latter type signatures for C'_0 and D'_0. The *CollectiveMatch* module can handle `MPI_Scatterv` similarly, but must scatter information on send type signatures instead.

For an `MPI_Alltoallv` collective, each place on the match layer receives arrays of send and receive type signatures, as Figure 5.6(a) illustrates with four processes. For a k-ary tree layout, p processes, and a collective that involves all processes, each place on the match layer can compare k^2 type signatures with its available data. With a hierarchical solution to execute type matching on all layers, the root would execute $\mathcal{O}(p^2)$ type signature comparisons, which would limit the scalability of the approach. Thus, the *CollectiveMatch* module uses intralayer communication to exchange type matching information for `MPI_Alltoallv` and `MPI_Alltoallw`. Figure 5.6(b) illustrates this exchange for the match layer and a layout as in Figure 5.5. Each place scatters its receive type signatures to all remaining places. Thus, the match layer performs an all-to-all exchange overall. Each place requires $\mathcal{O}(k \cdot p)$ time to distribute its type signatures (assuming that target places are idle), receive type signatures from other places, and to finally compare the type signatures.

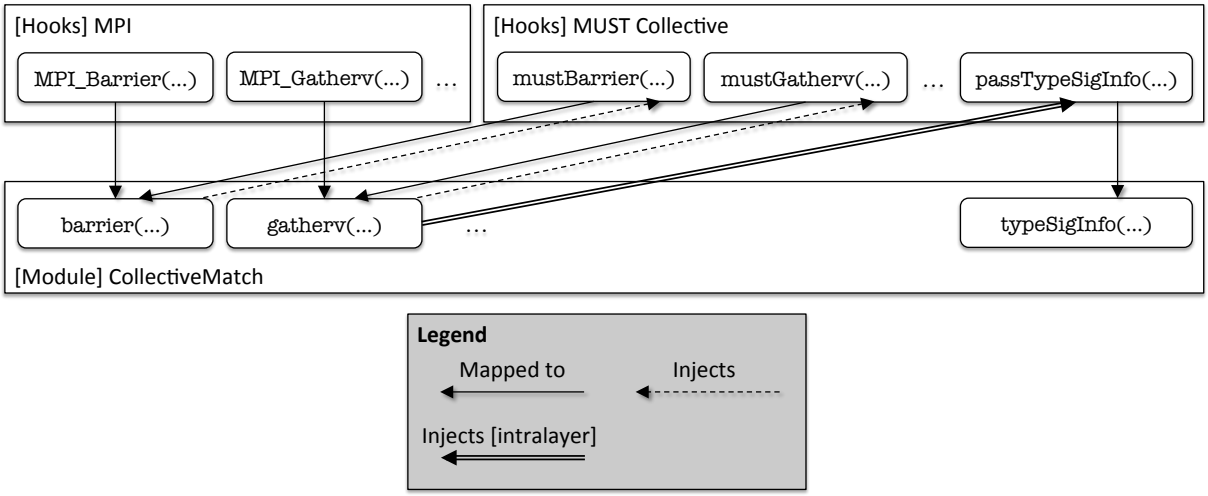


Figure 5.7: *A* module-hook *chart that illustrates how the proposed tool infrastructure abstraction enables the implementation of a* CollectiveMatch *module for distributed analysis of MPI collective operations.*

5.3.3 Module Design

Figure 5.7 illustrates the design of the *CollectiveMatch* module with a module-hook chart. The module uses one analysis for each collective, e.g., `barrier` for `MPI_Barrier`, and one tool internal hook to inject representative events for the respective collective, e.g., `mustBarrier`. When the *CollectiveMatch* module determines that all operations of a collective are consistent, it injects a new representative event with the hook. Otherwise, if an inconsistency exists, it forwards the original events, such as to detect multiple instances of the violation for detailed reporting. The hooks that the *CollectiveMatch* module uses include an additional argument that represents information on how many collective operations each event represents. This information supports the *CollectiveMatch* module when it determines whether all reachable operations of a collective arrived.

For collectives with non-transitive type matching rules, such as `MPI_Gatherv`, the `passTypeSigInfo` hook allows the *CollectiveMatch* module to forward type signature information with intralayer communication. Places that receive these injected events pass their information to the `typeSigInfo` analysis of the *CollectiveMatch* module. The module then determines to which collective the information belongs and applies the respective type signature comparisons for this collective.

The implementation of MUST varies this design such that it uses a condition module to split all collective operations into their respective send and receive transfers. This allows MUST to handle collectives with a root process without forwarding unused arguments (ones that are irrelevant on non-root processes and vice versa).

5.3.4 Data Structure

If event aggregations are successful, the *CollectiveMatch* module stores information on at most one active collective for each communicator, since GTI's order preserving event aggregation scheme would not enable an event that starts a second collective. However, if the module detects an inconsistency (and aborts the aggregation) or a timeout occurs on the place that hosts the module, the module may analyze multiple collectives for one communicator at a time. Thus, for each communicator, the module manages a single *active* collective for which an aggregation can still succeed and an array of collectives whose aggregations have been aborted. In addition, intralayer type matching information may arrive after the module perceived all operations of a collective. Thus, per communicator, an additional list manages all collectives for which intralayer type matching information is missing.

For k communicators, when a new collective operation arrives, the *CollectiveMatch* module requires

$\mathcal{O}(\log k)$ time to find the matching communicator. Afterwards, the module determines to which collective the event belongs. This requires up to l tests (one active and $l-1$ timed-out collectives). Each of these tests compares the channel identifier of the new operation against a data structure that resembles the queue tree from Section 4.5.2 (page 71). Thus, each test requires $\mathcal{O}(\log p)$ time for p application processes. If aggregations are successful on all layers, the comparison time is $\mathcal{O}(1)$, since channel identifiers will always identify a direct predecessor place of the current place. As stated previously, the actual correctness checks for an operation pair require $\mathcal{O}(1)$ time if the operations do not use arrays to specify irregular memory transfers, in which case the checks require $\mathcal{O}(p)$ time instead.

5.3.5 Time Complexities

In summary the analysis of a new operation requires either $\mathcal{O}(\log k + l \cdot \log p)$ (regular transfers) or $\mathcal{O}(\log k + l \cdot \log p + p)$ time (irregular transfers). If aggregations succeed on all layers of the layout, the *CollectiveMatch* module only analyzes a single collective per communicator and the queue tree structure has depth one. Thus, these time complexities become $\mathcal{O}(\log k)$ and $\mathcal{O}(\log k + p)$ respectively.

Table 5.2 summarizes all MPI-2.1 collective communications, the checks that apply to them, and information on how the *CollectiveMatch* module handles them. The "Types" Column specifies the scheme that compares type signatures. It includes "Local" for hierarchical comparisons on all layers of the tool (transitive scheme), "Local, Redundancy" for hierarchical comparisons that can use representatives for irregular communication operations due to redundant specifications in the operations, and "Intralayer" for intralayer-based exchanges of type signature information. The "GTI and MUST" column specifies both the time complexity that GTI requires to intercept a collective operation, communicate an event that represents the operation, and to trigger respective analyses on a tool place along with the analysis cost of the *CollectiveMatch* module. The GTI time cost is $\mathcal{O}(1)$ for all regular transfers where an event of constant size represents an operation and $\mathcal{O}(p)$ for all irregular transfers for which events include an array of size p (assuming idle target places in GTI's communication). These times exclude the potential impact of GTI's order preserving algorithm from Section 4.5 (page 69). The time complexity of the *CollectiveMatch* module uses the terms above and uses ∂ for the term $\log k + l \cdot \log p$ for brevity.

The "Communication Load (MPI)" column specifies the maximum number of data transfers that an operation of a collective can send or receive. As an example, for an `MPI_Reduce`, each operation sends one buffer and the root operation receives at least one buffer in addition, i.e., each operation has a load of $\mathcal{O}(1)$, since it transfers one or two buffers. For an `MPI_Gather`, the root operation receives one buffer from each process in the collective, i.e., the maximum load for this operation is $\mathcal{O}(p)$. The table shows that if ∂ is close to $\mathcal{O}(1)$, which holds for applications with few communicators and where aggregations succeed, the MUST and GTI handling costs match the communication load or is even lower than this load. Thus, the proposed approach can analyze the correctness of collective communications with similar costs as the original collective operations (irrespective of how an MPI library implements them). Scalable layouts must keep the fan-in constant when scale increases. Otherwise, places would analyze increasing amounts of collective operations, which would limit tool scalability in turn.

5.4 Single Execution Transition System

This thesis proposes a deadlock detection approach that combines: A transition system for low overhead analysis and graph-based deadlock detection to provide dependency information for detailed error reports. This section first introduces the rational that leads to this decision and then introduces a transition system that models the behavior of MPI operations. A specification of the transition system then enables a distributed implementation, which this section summarizes. Finally, a time complexity analysis for this distributed analysis defines costs for different types of MPI operations. The following section then introduces a wait-for-graph (WFG) module that captures a consistent state of the transition system and applies graph-based deadlock detection to it, such that the proposed approach can still provide detailed error reports.

			Checks				Communication	GTI and MUST	
			Call	Root	Op	Types	Load (MPI)	Comm. and Handling	*CollectiveMatch*
No Transfer			✓					$\mathcal{O}(1)$	$\mathcal{O}(\partial)$ $\partial = \log k + l \cdot \log p$
With Root	Transitive Matching	`MPI_Bcast`, `MPI_Reduce`	✓	✓	(✓)	Local	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\partial)$
		`MPI_Gather`, `MPI_Scatter`	✓	✓		Local	$\mathcal{O}(p)$	$\mathcal{O}(1)$	$\mathcal{O}(\partial)$
	Non-transitive Matching	`MPI_Gatherv`, `MPI_Scatterv`	✓	✓		Intralayer	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(\partial + p)$
Without Root	Transitive Matching	`MPI_Allreduce`, `MPI_Scan`, `MPI_Exscan`	✓		✓	Local	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\partial)$
		`MPI_Allgather`, `MPI_Alltoall`	✓			Local	$\mathcal{O}(p)$	$\mathcal{O}(1)$	$\mathcal{O}(\partial)$
	Redundancy Matching	`MPI_Allgatherv`, `MPI_Reduce_scatter`	✓		(✓)	Local, Redundancy	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(\partial + p)$
	Non-transitive Matching	`MPI_Alltoallv`, `MPI_Alltoallw`	✓			Intralayer	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(\partial + p)$

Table 5.2: *Time complexities to handle all MPI-2.1 collective communication operations with the proposed representative scheme and a comparison to the maximum communication load that these MPI operations involve (For p processes, k MPI communicators, and l collective operations in processing $\partial = \log k + l \cdot \log p$).*

5.4.1 Rationale

Timeout-based approaches to detect MPI deadlocks may report false positives and usually provide limited insight into the dependencies that cause a deadlock. Runtime detection approaches can overcome these drawbacks with a wait-for graph analysis or the use of transition systems.

Umpire as an early tool with an implicit WFG uses a graph search to detect deadlocks after analyzing each MPI operation. Per operation, this scheme can exhibit worst case analysis times of $\mathcal{O}(p^2)$ [67] and higher. The description of similar approaches such as MPIDD [64] also indicates the use of a graph search after analyzing each single MPI operation. An analysis of the wait-for dependency types that MPI operations can create [67, 69, 76] indicates that a single graph search for a deadlock requires $\mathcal{O}(p^2)$ time. The model in the study (AND$\oplus$OR model) can require additional graph nodes for some MPI operations, which can increase the search time. An empirical study with MPI applications uses heuristics to reuse knowledge from previous graph searches [67]. This scheme provides an approach that exhibits $\mathcal{O}(p)$ analysis time per MPI operation for several applications.

Since MPI point-to-point operations can have an actual cost of $\mathcal{O}(1)$ time, any deadlock analysis with an analysis cost of $\mathcal{O}(p)$ per operation limits scalability. This results from the fact that each process can issue MPI operations at a maximum rate that is about constant across scale. Thus, in worst case, the global rate at which an application issues MPI operations grows linearly with application scale. A distributed deadlock detection scheme must use its parallelism to cope with this linear increase, rather than with distributing $\mathcal{O}(p)$ time for the analysis of single operations. Consequently, a runtime deadlock detection approach must exhibit sub-linear analysis time per operation, in order to provide a basis for a distributed and scalable implementation.

Transition systems that mimic the semantics of MPI operations offer an alternative to graph-based approaches. Existing specifications [137, 142, 155] target the analysis of non-determinism in MPI applications with transition systems, in order to analyze all potential interleavings of an MPI application. While these specifications target non-determinism, informal approaches [69, 76] use an implicit transition system to model the behavior of MPI operations, but explicitly only consider a single execution. This approach allows a runtime tool to analyze whether a deadlock occurs in the interleaving that it observes during an application run. A centralized deadlock detection module of the MUST prototype [76] uses an informal transition system to analyze the conditions under which an MPI operation can return the control flow to an application process. A time complexity analysis of this approach along with an empirical study indicates that this approach can yield time complexities lower than $\mathcal{O}(p)$ per operation, where a wide range of applications have near constant or logarithmic increases with respect to scale. This approach provides the basis for a distributed implementation that could offer scalability, but lacks a formalism that underlines the soundness of the implicit model that analyzes MPI operations. Further, this approach uses a centralized module, i.e., has a scalability bottleneck.

Besides the lack of approaches for distributed implementations of low-cost transition systems for MPI, the use of a transition system comes with a disadvantage: A non-terminal state of a transition system, for which no further transition system rule applies, indicates the presence of a deadlock. This allows a tool to report all active MPI operations in a deadlock state. But, such a report offers no analysis as to which MPI operations cause the deadlock. Graph-based deadlock approaches, in turn, detect the core of the deadlock as their deadlock criterion and offer an approach to report and visualize it to the tool user. A study [124] details the types of error reports that a detailed graph-based deadlock detection enables.

Overall, this situation motivates a combination of a transition system for low overhead operation analysis with a graph-based detection to provide detailed error reports. In that case, a novel, distributed execution of the transition state could enable scalability.

5.4.2 Transition System

A transition system can capture an execution state of an MPI application and applies rules that determine whether an active MPI operation could return the control flow to the application. As an example, if an `MPI_Recv` operation is active in a state, it can return if the send operation that matches the receive is

Process 0	Process 1	Process 2
`MPI_Send`(to:1)	`MPI_Recv`(from:ANY)	`MPI_Send`(to:1)
	`MPI_Recv`(from:ANY)	
`MPI_Barrier`()	`MPI_Barrier`()	`MPI_Barrier`()
`MPI_Send`(to:1)	`MPI_Send`(to:2)	`MPI_Send`(to:0)
`MPI_Recv`(from:2)	`MPI_Recv`(from:0)	`MPI_Recv`(from:1)

Figure 5.8: *A series of MPI operations on three processes that includes a send-send deadlock, as well as the use of wildcard receives.*

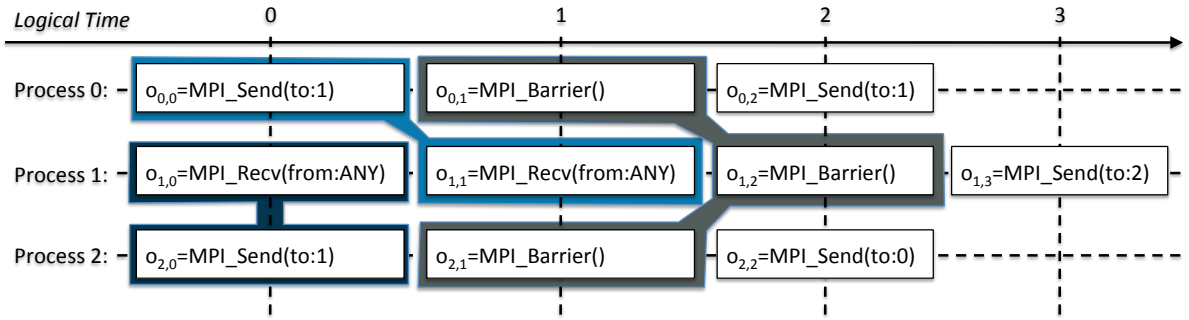


Figure 5.9: *A trace of MPI operations for the example from Figure 5.8 allows a transition system to analyze whether specific operations can unblock in a given execution state. Connected shades highlight matching operations.*

also active or was previously activated for the current state. If so, the transition system advances the state of the issuer of the receive to the next available operation. The subsequent definition for such a transition system is based on a previous publication [68]. The transition system uses a *trace* as input and utilizes a function that provides information on whether specific types of MPI operations require a condition to be met in order to return the control flow.

Let $P = \{0, 1, 2, \ldots, p-1\}$ be a finite set of process identifiers and let a finite sequence of MPI operations $t(i) = o_{i,0}, o_{i,1}, \ldots, o_{i,m_i}$ represent each process ($i \in P$). If no deadlock occurs, the last operation o_{i,m_i} is `MPI_Finalize` ($\forall i \in P$). A pair (i, j) where i is the process identifier (i.e., $i \in P$) and j the local and logical timestamp (i.e., $j \in \{0, \ldots, m_i\}$) subsequently refers to an operation in a sequence $t(i)$. The set of all MPI operations is:

$$\mathrm{Op} \stackrel{def}{=} \{(i, j) : i \in P, j \in \{0, \ldots, m_i\}\}$$

Figure 5.9 illustrates the traces of the three processes from the example application in Figure 5.8. This application issues two receive operations with wildcard sources on process 1, which match the two send operations of processes 0 and 2. Afterwards, all processes issue the collective operation `MPI_Barrier`, followed by a send operation. This application can deadlock since these standard mode send operations may block until a matching receive appears, which cannot be posted once that all processes enter their respective send operation. Each of the boxes in Figure 5.9 represents one of the operations and the trace ends at the send operations that can cause deadlock. Colored overlays connect matching operations and represent one of the two possible matches of the applications, i.e., $o_{1,0}$ could match $o_{0,0}$ instead of $o_{2,0}$. The trace must represent the matching decisions of the MPI implementation and the approach from Section 5.2 provides this information.

The transition system must consider whether an operation is blocking, i.e., waits until some condition is met, or nonblocking, i.e., will always return after some finite time. The Boolean-valued function

$b : \mathrm{Op} \to \{\bot, \top\}$ denotes this. The definition of b is:

$$b(i,j) \stackrel{def}{=} \begin{cases} & o_{i,j} \text{ is an } \texttt{MPI\_Send}, \texttt{MPI\_Ssend}, \texttt{MPI\_Recv}, \\ \top & : \texttt{MPI\_Probe}, \text{ a collective, or} \\ & \texttt{MPI\_Wait[any,some,all]}, \\ & \\ & o_{i,j} \text{ is an } \texttt{MPI\_Iprobe}, \texttt{MPI\_I[s,r,b]send}, \\ \bot & : \texttt{MPI\_\{B,R\}send}, \texttt{MPI\_Test[any,some,all]}, \\ & \text{or } \texttt{MPI\_Irecv} \end{cases}$$

For simplicity, b excludes persistent communication operations, since they have the same characteristics as nonblocking point-to-point operations. Further, `MPI_Sendrecv` can be represented as a series of operations such as the MPI standard suggests. Thus, the definition of b also excludes these send-receive operations. Also, the MPI standard lets implementations decide whether some operations, e.g., `MPI_Send`, use internal buffering or not. As a result, these operations may not be blocking, even though the definition of b indicates this. Subsequent sections consider these freedoms in detail. The trace in Figure 5.9 only uses blocking operations according to the definition of b.

Based on the definitions of the traces t_i ($i \in P$) and the blocking property b, the proposed transition system is $\mathcal{T} = (\mathrm{States}, \to_{\mathrm{ws}}, L_0)$, where States constitutes the state space of p-tuples $(l_0, \ldots, l_{p-1})$ and where $l_i \in \{0, 1, \ldots, m_i\}$ for $i \in P$. Each state represents the logical timestamps of the currently active MPI operations at an execution state of the application. The initial state $L_0 = (0, \ldots, 0)$ activates the first operation of all processes and the transition relation $\to_{\mathrm{ws}} \subseteq \mathrm{States} \times \mathrm{States}$ is the smallest binary relation on States that satisfies the following rules[2]:

(1) $o_{i,j}$ is a nonblocking operation:

$$\frac{b(i,j) = \bot \wedge l_i = j}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \stackrel{\mathrm{nb}}{\to}_{\mathrm{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

(2) $o_{i,j}$ is a send/receive/probe operation where $o_{k,n}$ is the matching receive/send/send operation:

$$\frac{l_i = j \wedge l_k \geq n}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \stackrel{\mathrm{p2p}}{\to}_{\mathrm{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

(3) $C = \{(i_0, j_0), (i_1, j_1), \ldots, (i_n, j_n)\}$ is a complete set of matching collective operations (i.e., each process in the process set associated with the collective is present):

$$\frac{(i,j) \in C \wedge l_i = j \wedge \forall (k,n) \in C : l_k \geq n}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \stackrel{\mathrm{coll}}{\to}_{\mathrm{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

(4) $o_{i,j}$ is a completion operation that uses MPI requests that are associated with nonblocking send/receive operations $o_{i,j_0}, o_{i,j_1} \ldots, o_{i,j_x}$ ($\forall k \in \{0, \ldots, x\} : j_k < j$):

(I) $o_{i,j}$ is an `MPI_Waitany` or `MPI_Waitsome` and the send/receive operation o_{i,j_y} for some $y \in \{0, \ldots, x\}$ is matched with the receive/send operation $o_{k,n}$:

$$\frac{l_i = j \wedge l_k \geq n}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \stackrel{\mathrm{any}}{\to}_{\mathrm{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

(II) $o_{i,j}$ is an `MPI_Wait` or `MPI_Waitall` operation and for all $y \in \{0, \ldots, x\}$, the send/receive o_{i,j_y} is matched with the receive/send operation o_{k_y,n_y}:

$$\frac{l_i = j \wedge \forall y \in \{0, \ldots, x\} : l_{k_y} \geq n_y}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \stackrel{\mathrm{all}}{\to}_{\mathrm{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

[2]Rules (1)-(4) and subsequent transition examples use labels for the transitions to indicate the type of the executed operation, e.g., $\stackrel{\mathrm{nb}}{\to}_{\mathrm{ws}}$ for Rule (1).

Rule (1) reflects that nonblocking operations always return after a finite amount of time, whereas the other rules define when a process is allowed to return from a blocking operation. For a process with an active point-to-point operation—rule (2)—the process may advance to the next operation if the matching operation is also active. This rule also includes the use of `MPI_Probe`, which only differs from a blocking receive operation in its consequences for point-to-point matching, since it does not receive a message. Similarly, for processes that are active in a collective—rule (3)—a process may advance to the next operation if all processes that belong to the collectives process group activated their participating operation. Note that if a deadlock manifests, a matching operation may not exist, e.g., for $o_{0,2}$ in Figure 5.9. The presence of a matching operation is a premise for rules (2)–(4).

The rules do not enforce an order in which processes activate the next operation, which represents the semantics of MPI and reduces the need for synchronization, which simplifies a distributed implementation. In particular rule (2) allows receiver processes in point-to-point communications to advance to the next operation while the sender is still active in the send. In such a case the sender would have communicated the complete message buffer to the receiver, while the sender still handles local computations, e.g., frees a temporary buffer. Rule (4) handles blocking completion operations, i.e., `MPI_Wait[any,some,all]`. For *any*/*some* completions it requires that a matching and active operation exists for at least one associated nonblocking communication. The two types of operations only differ in that MPI may complete multiple communications in `MPI_Waitsome`, which influences the trace of operations that MUST records, but not the transition system rules. For *all* completions—`MPI_Waitall`—a matching and active operation must exist for all associated nonblocking operations.

The executions of the transition system $\mathcal{T}$ are generated by starting in the initial state L_0 and then repeatedly moving to a successor state according to the transition relation $\rightarrow_{\text{ws}}$ until no further rule can be applied. For the example in Figure 5.9 a possible execution of the transition system is: $(0,0,0) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (0,0,1) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (0,1,1) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (0,2,1) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (1,2,1) \overset{\text{coll}}{\rightarrow}_{\text{ws}} (1,2,2) \overset{\text{coll}}{\rightarrow}_{\text{ws}} (2,2,2) \overset{\text{coll}}{\rightarrow}_{\text{ws}} (2,3,2)$. To illustrate the preconditions of the rules, consider the state $(0,0,1)$: In this state, rule (2) cannot again be applied to $o_{2,0}$, since $l_2 \neq 0$, i.e., this operation is not the current operation of process 2. Also rule (2) is not applicable to $o_{0,0}$, even though it is the active operation of process 0, since the matching operation $o_{1,1}$ is not active in this state (the state does not satisfy the second part of the precondition of rule (2)). As a last example, rule (3) is not applicable to $o_{2,1}$. While this operation is active on process 2, both matching operations $o_{0,1}$ and $o_{1,2}$ are not active in this state, i.e., the state does not satisfy the second part of the precondition of rule (3).

5.4.3 Deadlock Criterion

The transition system rule for collective operations also applies to other operations that must be executed collectively, e.g., `MPI_Comm_dup`. The only exception is the last operation m_i in the trace ($\forall i \in P$) that represents `MPI_Finalize`. No transition system rule applies to this operation, such that it serves as a well-defined terminal state (if reachable). Consequently, the transition system always arrives in a terminal state $(l_0, \ldots, l_{p-1})$, where either $l_i = m_i$ for all $i \in P$ or $l_i < m_i$ for some i. In the former case, the transition system could analyze all operations in the trace and arrived at the `MPI_Finalize` operations for which no rule is applicable, i.e., no deadlock exists in the analyzed trace. In the latter case, where some $l_i < m_i$ exists, the transition system revealed a deadlock. Thus, the transition system itself is sufficient to detect deadlocks and provides information on the terminal state, i.e., the operations that are active during the deadlock.

Only one final state exists even though the transition system is nondeterministic, e.g., for the example in Figure 5.9 both $(0,0,0) \rightarrow_{\text{ws}} (0,0,1)$ and $(0,0,0) \rightarrow_{\text{ws}} (0,1,0)$ are choices for the first transition. However, each rule that allows an application of $\rightarrow_{\text{ws}}$ never disables any transition that could have been applied at a previous state. That is, if a rule can increment l_k at state $(l_0, \ldots, l_{k-1}, l_k, l_{k+1}, \ldots, l_{p-1})$, then all states $(l'_0, \ldots, l'_{k-1}, l_k, l'_{k+1}, \ldots, l'_{p-1})$ with $l'_i \geq l_i$ ($\forall i \in P \setminus \{k\}$) still allow the application of this rule. Thus, a terminal state must exist since the traces have finite length.

To combine graph-based deadlock detection with the transition system in this section, the latter must

Process 0	Process 1	Process 2
`MPI_Send`(to:1)	`MPI_Recv`(from:ANY)	
`MPI_Reduce`(root=1)	`MPI_Reduce`(root=1)	`MPI_Reduce`(root=1)
	`MPI_Recv`(from:ANY)	`MPI_Send`(to:1)

Figure 5.10: *A series of MPI operations on three processes that enables an* unexpected match*. For MPI implementations that implement* `MPI_Reduce` *without global synchronization, the send operation of process* 2 *can match the first wildcard receive of process* 1*.*

provide wait-for dependencies for a current state. The current state of the transition system may include operations for which a transition rule can be applied, i.e., which can unblock in the current state. These operations must not create any wait-for conditions as a result. Thus, a process $i \in P$ is blocked in a current state $S = (l_0, \ldots, l_i, \ldots, l_{p-1})$ if no $(S, S') \in \rightarrow_{\text{ws}}$ with $S' = (l_0, \ldots, l_i + 1, \ldots, l_{p-1})$ exists.

The state $(2, 3, 1)$ for the example trace in Figure 5.9 illustrates this definition. Processes 0 and 1 arrived in the send operations in which they will deadlock, while process 2 still uses the `MPI_Barrier` operation as its active operation. A wait-for graph for this state uses dependencies for processes 0 and 1 since no transition is enabled for them, but does not use dependencies for process 2, since a transition applies to this process. A WFG model for MPI [69] allows a representation of the blocked MPI operations as a dependency graph. The graph would include that process 0 waits for process 1, due to its unmatched `MPI_Send` operation and that process 1 waits for process 2, due to its also unmatched `MPI_Send` operation. This graph does not indicate deadlock for the AND⊕OR model [69]. Afterwards, when the transition system arrives in the terminal state $(2, 3, 2)$, all operations provide wait-for dependencies for graph-based deadlock detection, since no transition is enabled for any process. These wait-for dependencies form a cycle (a necessary and sufficient deadlock criterion in this situation) and reveal the deadlock similarly to the terminal state of the transition system, but with the ability to provide enhanced outputs.

5.4.4 Freedoms of the MPI Standard

The MPI standard grants implementations freedoms as to whether specific types of MPI operations are blocking or not. This includes operations such as `MPI_Send` and collective communication operations such as `MPI_Reduce`. For the first operation, an implementation can buffer the payload of the send transfer and apply a nonblocking implementation, while for the second operation, an implementation may avoid a synchronization across all processes of the collective. The definition of b, however, is fixed and assumes the strictest interpretation of the MPI standard in such cases. This allows the transition system to arrive in a terminal state where the application continues its execution. In such a case the tool detects a deadlock that would manifest for stricter MPI implementations, i.e., a portability defect.

Wildcard receives in combination with an operation that may or may not be blocking can cause *unexpected* matches. Figure 5.10 illustrates an example that can yield an unexpected match and that is free of deadlock. If the `MPI_Reduce` operation is synchronizing, the send-receive pairs before and after the collective will match respectively. However, if the MPI implementation allows the collective operation of process 2 to return before process 1 enters the collective, then the send of process 2 can match the first receive of process 1. In that case, the trace specifies that the first wildcard receive of process 1 matches the send of process 2. However, the transition system will not be able to apply any rule to its initial state, since: For process 0 no matching receive is available, for process 1 its matching send is not active in the state, and process 2 is the only process active in its collective.

An unexpected match for a wildcard receive o_{i_1,j_1} is a terminal state S of the transition system in which a send operation o_{i_2,j_2} exists that is active in S and that could match the active operation o_{i_1,j_1}, while the MPI implementation returns information on a match with another send that is not active in S. The transition system must weaken its definition of b for such unexpected matches and the above definition allows an implementation to detect such scenarios. The subsequent distributed implementation does not adapt the definition of b for unexpected matches. However, while the transition system arrives in a

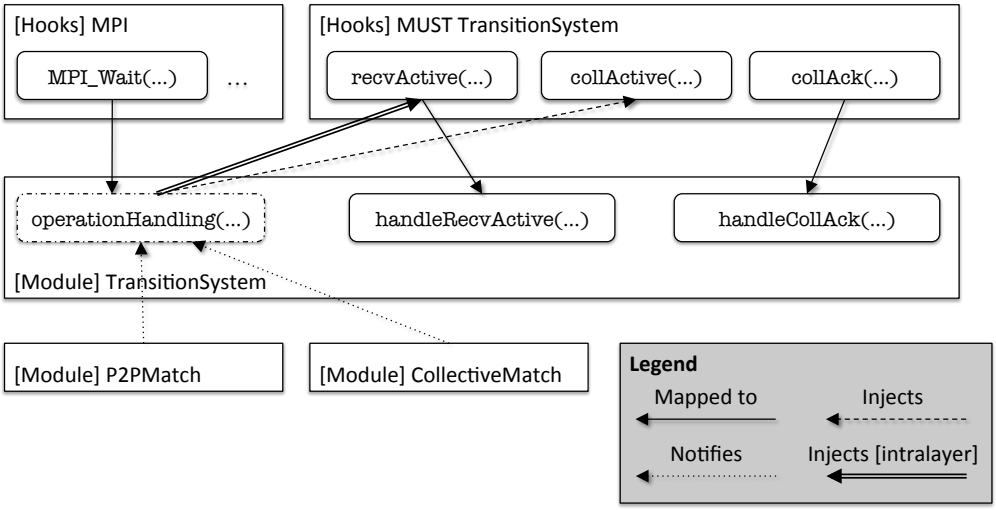


Figure 5.11: *A* module-hook *chart that illustrates how the proposed tool infrastructure abstraction enables the implementation of a* TransitionSystem *module for a distributed transition system implementation.*

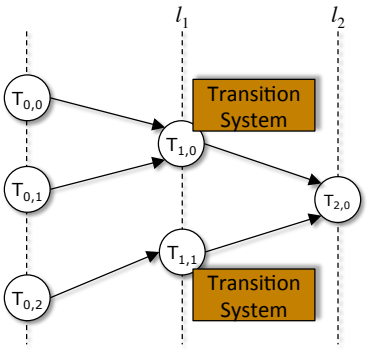
terminal state in such a situation, the graph-based deadlock detection that Section 5.5 introduces will not report a deadlock.

5.4.5 Distributed Implementation

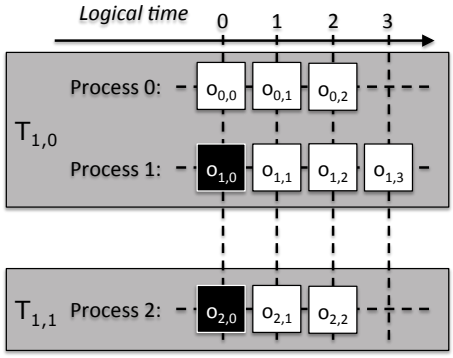
A decomposition of the transition system state provides a basis for a distributed implementation of the transition system. The MUST prototype implements this distributed transition system with the *TransitionSystem* module. Rule (2) requires information on matching point-to-point operations and whether these operations are active in a current state. Since the former information is directly available on the match layer, the MUST prototype places the *TransitionSystem* module onto the match layer too. Similarly to the domain decomposition in Section 5.2, the module distributes a state $S = (l_0, l_1, \ldots, l_{p-1})$ such that if processes $i, i+1, \ldots, i+k \in P$ send events along the primary communication direction to a place of the match layer, then this place handles the components $l_i, l_{i+1}, \ldots, l_{i+k}$ of S.

Since both the *P2PMatch* and the *CollectiveMatch* modules run on the match layer, these modules can provide information on matching point-to-point and collective operations to the *TransitionSystem* module. Figure 5.11 presents a module-hook chart for the *TransitionSystem* module. The fine-dashed lines that originate at the *P2PMatch* and *CollectiveMatch* modules highlight the information flow from these modules to the *TransitionSystem* module—a callback mechanism and module dependencies implement it. The analysis `operationHandling` (dashed-dotted line) summarizes multiple analyses and callbacks to simplify the figure. A subsequent description highlights the behavior of this handling. Analysis-hook mappings onto all MPI completion calls—all versions of `MPI_Test` and `MPI_Wait`—provide the *TransitionSystem* module both information for the management of nonblocking point-to-point operations and on operations that belong to the trace of operations. Thus, the callbacks that the *P2PMatch* and the *CollectiveMatch* modules invoke add point-to-point and collective operations to the trace, while analysis mappings add completion operations to the trace.

Figure 5.12(a) illustrates a tool layout for three application processes. Let this layout apply to the example series of MPI operations from Figure 5.8. For this layout, Figure 5.12(b) illustrates that each *TransitionSystem* module instance on the match layer receives a subset of the overall operation trace. Additionally, each module instance covers a subset of the transition system state, i.e., $T_{1,0}$ covers the states of processes 0 and 1, while $T_{1,1}$ covers the state of process 2. The send-receive operation pair $o_{1,0}$ and $o_{2,0}$ (highlighted in white on black) illustrates the need to exchange information between the places of the match layer. On place $T_{1,0}$, the *TransitionSystem* module perceives information on operation $o_{1,0}$



(a) A tool layout for the three application processes in Figure 5.8 with the *TransitionSystem* module on the match layer.



(b) Places $T_{1,0}$ and $T_{1,1}$ receive parts of the overall trace of operations with the layout in (a). Matching operations such as $o_{1,0}$ and $o_{2,0}$ are distributed over both places.

Figure 5.12: *The data distribution that results with a placement of the* TransitionSystem *module requires the exchange of state information.*

and on its matching operation $o_{2,0}$, due to the intralayer communication of the *P2PMatch* module, which forwards information on $o_{2,0}$ to $T_{1,0}$. However, in order to apply rule (2), the module instance also requires information on the current transition system state of process 2, which is hosted on place $T_{1,1}$. Similarly, place $T_{1,1}$ neither perceives information on the state of process 1 nor on the operation that matches $o_{2,0}$. The *TransitionSystem* module uses the following hooks to exchanges information on rule premises for a global state $(l_0, l_1, \ldots, l_{p-1})$:

- `passSend`: [intralayer-direction] (from Figure 5.3) Passes information on an active send operation o_{i_1,j_1} hosted on place T to the place T', which hosts the matching receive operation o_{i_2,j_2}, includes the timestamp of the send (provides T' with the information that o_{i_1,j_1} and o_{i_2,j_2} match, as well as that the precondition of rule (2) is satisfied for i_2, i.e., that $l_{i_1} \geq j_1$);
- `recvActive`: [intralayer-direction] Informs place T hosting an active send operation o_{i_1,j_1} that the matching receive operation o_{i_2,j_2} is now also active (provides T with the information that o_{i_1,j_1} and o_{i_2,j_2} match, as well as that the precondition of rule (2) is satisfied for i_1, i.e., that $l_{i_2} \geq j_2$);
- `collActive`: [primary-direction] For a collective, if a place T can receive information on the participating collective operations $o_{i_1,j_1}, o_{i_2,j_2}, \ldots, o_{i_n,j_n}$ along the primary communication direction: If all of these operations are active in the current state ($\forall x \in \{i_1, i_2, \ldots, i_n\} : l_x \geq j_x$), T sends this message towards the TBON root; and
- `collAck`: [broadcast-direction] If the root of the TBON determines that all processes $k_0, \ldots, k_n$ that belong to a collective have activated their participating operation (match layer places provided the respective `collActive` events), it broadcasts this message towards the match layer (notifies all places that processes $k_0, \ldots, k_n$ satisfy the precondition of rule (3)).

When the *TransitionSystem* module registers a callback with the *P2PMatch* module—to add point-to-point operations to its trace—it also modifies the behavior of the latter module. The default handling of the *P2PMatch* module passes information on send operations with the intralayer hook `passSend`, whenever it receives information on a new send operation. However, with the presence of the *TransitionSystem* module, the *P2PMatch* module only forwards this information once that the respective send operation became active. The `passSend` hook then provides all the information that the receiver place requires to apply rule (2) to the matching receive call—if the receive is blocking and only after it became active. The intralayer hook `recvActive` provides similar data to *TransitionSystem* module instances that host receive operations. The `passSend` hook includes the logical timestamp of the send operation, such that a subsequent *recvActive* hook can include this timestamp. Receivers of *recvActive* events

use the send operation timestamp that these events provide to identify the operation to which rule (2) is applicable. Figure 5.11 illustrates that the `handleRecvActive` analysis of the *TransitionSystem* module is mapped to the `recvActive` hook to perceive premise information for point-to-point receive operations.

Each instance of the *TransitionSystem* module handles traces for one or multiple processes. In order to apply rule (3), module instances must determine whether all processes in the process group of a collective activated their participating operations. Thus, instances of the *TransitionSystem* module must exchange information in order to assure that the premise of rule (3) holds. The *TransitionSystem* module injects an event with the `collectiveActive` hook (primary communication direction) when all the processes that it manages are active in a collective (or the respective sub-group of these processes if the communicator of the collective does not include all processes). An aggregation module combines these events from individual *TransitionSystem* module instances. A further module on the root then checks whether all processes in a collective's process group activated their participating operations. If so, this module injects an event with the `collAck` hook (broadcast direction), which notifies all *Tranisition-System* module instances that the premise of rule (3) is valid for the respective collective. Appendix B.4 (page 170) details these additional modules and their analysis-hook mappings.

5.4.6 Operation Processing

Figure 5.11 uses the analysis `operationHandling` that combines analyses, callbacks, and a progress loop. Analysis-hook mappings provide information on MPI completion operations, the callbacks provide information on point-to-point and collective operations from the *P2PMatch* and *CollectiveMatch* modules, and the progress loop checks whether transition system rules are applicable, or whether events must be injected to forward information to other places. The handlers in Figure 5.13 details how the *TransitionSystem* module reacts to specific events and when it injects an event in response. It uses o to refer to an operation and represents them as C++-like objects, i.e., "o." selects an attribute or member of the object. The attribute $o.l$ is the timestamp of the operation, $o.l_s$ the timestamp of a matching send operation, and $o.\text{active}$ expresses whether the operation is active in the current transition system state. For an active operation o of a process $i \in P$, the additional attribute $o.\text{canAdvance}$ specifies whether process i can take the transition from l_i to $l_i + 1$, i.e., whether any transition rule applies to the operation. The handlers set this attribute to $\top$ for $o_{i,j}$ if:

- $b(i, j) = \bot$,
- A `collectiveAck` has arrived for collective $o_{i,j}$,
- A `passSend` has arrived for the active receive $o_{i,j}$,
- A `recvActive` has arrived for the active send $o_{i,j}$, or
- For the completion operation $o_{i,j}$ that is associated with the nonblocking send/receive operations $o_{i,j_0}, o_{i,j_1}, \ldots, o_{i,j_x}$ ($\forall k \in \{0, \ldots, x\} : j_k < j$):
  - $o_{i,j}$ is either an `MPI_Waitany` or an `MPI_Waitsome` and $\exists y \in \{0, \ldots, x\}$ such that $\text{canAdvance}(o_{i,k_y}) = \top$, or
  - $o_{i,j}$ is an `MPI_Wait` or `MPI_Waitall` and $\forall y \in \{0, \ldots, x\}$ $\text{canAdvance}(o_{i,k_y}) = \top$.

The handlers in Figure 5.13 exclude the handling of blocking completions, but the above rules detail the conditions under which a transition system rule applies to such an operation.

Figure 5.14 illustrates a possible processing order to handle the highlighted point-to-point operations of Figure 5.12(b), which are the receive $o_{1,0}$ on place $T_{1,0}$ and the send $o_{2,0}$ on place $T_{1,1}$. The figure highlights the communication between the two places to exchange information on transition rule premises. Place $T_{1,1}$ receives and handles its operation before $T_{1,0}$ and immediately activates it. Thus, it uses the handlers `newOp` and `activate` from Figure 5.13. When $o_{2,0}$ becomes active, its host module injects an event with the `passSend` hook to notify $T_{1,0}$ of the activation of the send operation. Place

Function `newOp(`o`)`

```
1 o.l := nextTimestamp()
2 o.active := ⊥
3 if o.isBlocking() then
4 |   o.canAdvance := ⊥
5 else
6 |   o.canAdvance := ⊤
```

Function `activate(`o`)`

```
1 o.active := ⊤
2 if o.isSend() then
  |   /* P2PMatch module executes this in the actual implementation */
3 |   communicate::passSend(..., o.l)
4 if isLastInactiveCollectivOperationOnPlace(o) then
5 |   communicate::collActive(o.comm)
6 if o.isRecv() ∧ o.hasMatchingSend() then
7 |   communicate::recvActive(o.l_s)
```

Function `handleCollAck(`communicator`)`

```
1 o_0, ..., o_k := findActiveOpsInTrace(communicator)
2 for o_i in o_0, ..., o_k do
3 |   o_i.canAdvance := ⊤
```

Function `handlePassSend(...,`l_s`)`

```
1 if hasMatchingRecv(...) then
2 |   recv := findMatchingRecv(...)
3 |   recv.l_s := l_s
4 |   if recv.active then
5 |   |   communicate::recvActive(recv.l_s)
6 else
7 |   storeSendForMatching(..., l_s)
```

Function `handleRecvActive(`l_s`)`

```
1 send := findOpInTrace(l_s)
2 send.gotRecvActive := ⊤
3 send.canAdvance := ⊤
```

Figure 5.13: *The handlers of the* TransitionSystem *module that enable an exchange of state information between instances of the module.*

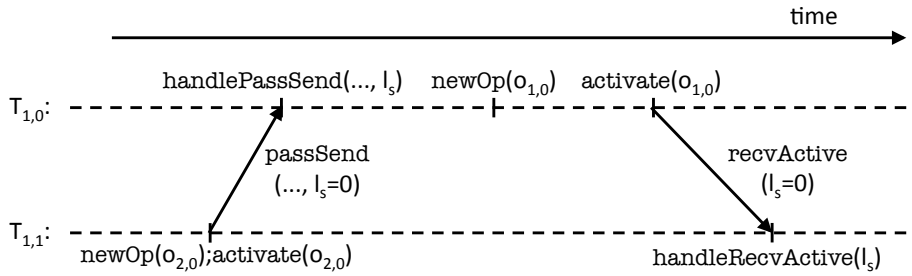


Figure 5.14: *An information exchange with the* `passSend` *and* `recvActive` *hooks for operations* $o_{1,0}$ *and* $o_{2,0}$ *from Figure 5.12(b). The illustration highlights how instances of the* TransitionSystem *module exchange state information for MPI point-to-point operations.*

$T_{1,0}$ receives and handles this event with the `handlePassSend` handler. The timing in the example lets this information arrive on place $T_{1,0}$ even before it receives operation $o_{1,0}$. Thus, the *P2PMatch* module on this place stores information on the send operation and its timestamp in its matching structures. Afterwards, when $T_{1,0}$ receives, handles, and activates operation $o_{1,0}$—including an update of the wildcard receive source to determine the matching decision of the MPI implementation—the place injects an event with the `recvActive` hook to notify $T_{1,1}$ of the activation of the receive operation. Place $T_{1,1}$ then handles this notification with the `handleRecvActive` handler and uses the timestamp of the message to identify the send operation in question. Place $T_{1,0}$ sets $\mathrm{canAdvance}$ to $\top$ for $o_{1,0}$ immediately when it activates the operation, since it observes the `passSend` event for this operation beforehand. Place $T_{1,1}$ sets $\mathrm{canAdvance}$ to $\top$ for $o_{2,0}$ when it handles the `recvActive` event. Afterwards, the progress loop on these *TransitionSystem* module instances can advance to the next operation to continue the execution of the transition system.

5.4.7 Data Structure

A complete operation trace, even for a fixed number of processes, can exceed the available main memory on a compute node. However, the *TranisitionSystem* module can remove processed operations from the trace. This includes all operations with timestamps that are below the current timestamp of their issuer process in the state vector. The only exception are nonblocking point-to-point communications that can be referenced by a latter completion operation. A reference count mechanism in combination with mappings to all MPI completion calls ensures that the *TranisitionSystem* module removes nonblocking communication operations when the MPI implementation completed them and no further completion operation in the trace references them. As a consequence, if module instances process operations faster than new operations arrive on places, the trace requires a bounded amount of memory.

The *TranisitionSystem* module uses a trace data structure that maps the timestamp of an operation to the object that represents it. This mapping simplifies access to operations during processing. The data structure in the MUST prototype implements addition to the trace and access to an element in the trace with $\mathcal{O}(\log l)$ time for a trace size of l. This allows the module to use timestamps as operation identifiers within the interfaces to the *CollectiveMatch* and *P2PMatch* modules. The use of a pointer along with a linked list data structure could reduce the access and management time to $\mathcal{O}(1)$ in a future implementation.

An additional map stores information on uncompleted nonblocking point-to-point communication operations, which MPI identifies with a request in completion operations. Callbacks from the *P2PMatch* module add new requests of nonblocking point-to-point communications to the map when the application initiates them. Analysis-hook mappings of the *TransitionSystem* module remove these entries when a subsequent completion operation completes them. To associate a blocking completion operation, with r requests, requires $\mathcal{O}(r \cdot \log q)$ time to associate the completion with each of the r nonblocking com-

	Point-to-point operation	Collective operation	Completion operation r requests
Create operation	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(r \cdot \log q)$
Add/remove to/from trace	$\mathcal{O}(\log l)$	$\mathcal{O}(\log l)$	$\mathcal{O}(\log l)$
Activate	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Communication	$\mathcal{O}(\tau)$	$\mathcal{O}(\tau \cdot \log p)$	–
Update $\mathrm{canAdvance}$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(r)$
Delete operation	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(r)$
Total	$\mathcal{O}(\tau + \log l)$	$\mathcal{O}(\tau \cdot \log p + \log l)$	$\mathcal{O}(\log l + r \cdot \log q)$

Table 5.3: *Time complexities within the operation handling of the* TransitionSystem *module for p processes, trace length of a process l, q open nonblocking operations, and a communication response time of τ.*

munications out of a total of q open nonblocking communications. The number of uncompleted communication operations q is independent of the actual analysis state of the transition system and matches the number of uncompleted point-to-point operations on the application process at the time when it issued the completion.

5.4.8 Time Complexities

Table 5.3 summarizes the time complexities of the *TransitionSystem* module for point-to-point, collective, and completion operations. For a point-to-point operation—irrespective of whether it is blocking or not—the module uses an object of $\mathcal{O}(1)$ memory size to represent the operation and adds it to the trace with $\mathcal{O}(\log l)$ time for a trace of length l. When the progress loop of the module activates the operation, it can inject a single event (either with `recvActive` or `passSend`) that has a payload of $\mathcal{O}(1)$ and that thus requires $\mathcal{O}(1)$ communication time (assuming a responsive target place). The *P2PMatch* and *CollectiveMatch* modules of the previous section also used communication to handle their events, but their event handling did not require a response message to enable progress. However, the *TransitionSystem* module may have to wait until a response arrives before it can apply a transition system rule. The previous sections assumed idle target places for an $\mathcal{O}(1)$ communication time, where the MUST prototype employs event buffering to usually ensure this cost even if target places are not idle. Due to the need of a response event, this section assumes a *communication cost* τ to represent the response time of a target place. This time depends on the load of the place and its processing order. The *TransitionSystem* module updates the $\mathrm{canAdvance}$ attribute of the currently active operation with $\mathcal{O}(1)$ time when a response arrives for a point-to-point operation. Deleting the operation and removing it from the trace requires $\mathcal{O}(\log l)$. In summary, the handling of a point-to-point operation requires $\mathcal{O}(\tau + \log l)$. With a short trace and responsive places, this cost can match the time that an MPI implementation requires for such an operation.

The analysis cost for collective operations only differs in the response time that requires that $\log p$ places process and respond to a `collActive` message. More exactly $\log_k p$ for a layout with a constant fan-in of k. This yields a total time complexity of $\mathcal{O}(\tau \cdot \log p + \log l)$ (assuming a similar response time for all places) that can match the actual cost of collectives that use a hierarchic implementation. Besides the influence of the response time and the trace length, MPI implementations that use specific multicast/-broadcast or synchronization network capabilities [4] can outperform the *TransitionSystem* module.

The total cost of a completion operation with r requests is $\mathcal{O}(\log l + r \cdot \log q)$ for a blocking completion and $\mathcal{O}(r \cdot \log q)$ for a nonblocking completion, which only updates the map of open nonblocking point-to-point communication operations. The time complexity for both types of completions includes the lookup time for each operation associated with every request, which is $\mathcal{O}(r \cdot \log q)$. Blocking completion operations require no communication time since the handling time of the respective point-to-point operations already includes this overhead. However, to check whether the progress loop can set $\mathrm{canAdvance}$

to true requires $\mathcal{O}(r)$ time to check the state of up to r communications. Again the handling cost for blocking completions may exceed the handling time of an MPI implementation, but is still independent of the application scale.

In summary, these complexities provide promising characteristics for a distributed transition system implementation. Especially, these costs do not include any $\mathcal{O}(p)$ complexities that would limit scalability. This distribution scheme combines low analysis costs per MPI operation as in a previous centralized deadlock detection [76] approach with an informal transition system. However, while the previous approach could not scale due to its centralized analysis, the proposed scheme distributes the operation processing.

5.5 Deadlock Detection

The transition system implementation of the last section suffices to detect deadlock in MPI applications. If each process has an active operation, no transition rule is applicable, and the *TransitionSystem* module instances processed and perceived all events from the `passSend`, `recvActive`, `collActive`, and `collAck` hooks, then the module arrived at a terminal state. If at least one operation in a terminal state is not `MPI_Finalize` and no unexpected match exists, then the transition system revealed a deadlock.

This section proposes a combination of graph-based deadlock detection with a transition system to provide more detailed error reports. The MUST implementation uses the *WfgManager* module for the graph search and interfaces it with the *TransitionSystem* module, such that is can analyze any of its intermediate states. The actual detection uses the AND⊕OR model [69] with a generalization [76] that transforms AND-OR wait for semantics [14, 106, 172] into the former model. The *WfgManager* module is centralized—runs on a single place—and uses a timeout to initiate deadlock detection, since the time complexity of the detection prohibits continuous detection. This section presents the design of the *WfgManager* module with its synchronization scheme, which assures that the input for the graph-based deadlock detection is consistent. A second interface then allows instances of the *TransitionSystem* module to describe the wait-for dependencies of all active operations to which no transition rule applies in the current state. This input then forms the basis for graph-based deadlock detection.

5.5.1 Consistent State

The previous section introduced the relation between a transition system state and a graph-based deadlock detection: Given a transition system state $l_0, l_1, \ldots, l_i, \ldots, l_{p-1}$, a process i has wait-for dependencies in this state if no rule applies that could advance l_i to $l_i + 1$. In a centralized transition system implementation, all information to evaluate this property is immediately available. However, for the distributed implementation of MUST, the overall state is distributed across multiple module instances. When one instance applies a transition rule, it communicates events to notify other instances of the state change. Inconsistent global states exist, since this communication is not instantaneous, i.e., if a request for wait-for information arrives while events of the `passSend`, `recvActive`, `collActive`, and `collAck` hooks are in transit, then a module instance may not be able to evaluate the above property correctly. Thus, a synchronization scheme must ensure that all relevant events are handled before any module reports wait-for dependencies.

To illustrate an inconsistent state, consider the point-to-point operation handling in Figure 5.14. If a request for wait-for data arrives on $T_{1,1}$ after it activated $o_{2,0}$ and before it received the `recvActive` event from $T_{1,0}$, then it would report a wait-for dependency for $o_{2,0}$ since the place can't apply a transition rule to this operation. When $T_{1,0}$ handles the request for wait-for information itself, the wait-for dependency reported by $T_{1,1}$ is correct if $T_{1,0}$ did not activate $o_{1,0}$ yet. However, if $T_{1,0}$ already activated $o_{1,0}$, then $T_{1,1}$ must not report any wait-for dependencies for $o_{2,0}$, since rule (2) is applicable to this operation then.

Figure 5.15 presents a module-hook chart for the analyses of the *WfgManager* module, as well as for the analyses of the *TransitionSystem* module, which exchange wait-for data for the graph-based deadlock

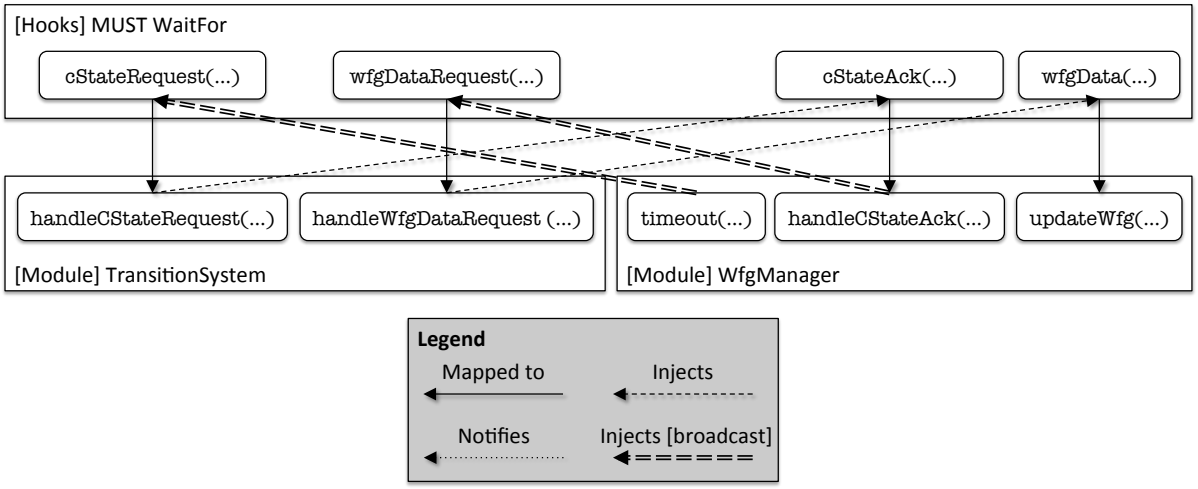


Figure 5.15: *A* module-hook *chart that illustrates how the proposed tool infrastructure abstraction enables the synchronization for a consistent state of the* TransitionSystem *module, as well as the forwarding of WFG data for a centralized graph search.*

detection and provide synchronization for a consistent global transition system state. The hooks in the figure provide the necessary tool internal communication for these tasks. A configurable timeout on the *WfgManager* module triggers deadlock detection. A callback from a utility module (*TSRequestManager* in Appendix B.4 on page 170) resets the timeout whenever all processes issue a collective communication to avoid unnecessary deadlock detections. When a timeout occurs, the *WfgManager* module first injects an event with the `cStateRequest` hook (broadcast direction) to initiate the synchronization that ensures a consistent state. When a *TransitionSystem* module instance handles this event with the `handleCStateRequest` analysis it:

(i) Locks its portion of the global state, i.e., disables the use of any transition rules;

(ii) Determines all active send operations for which no `recvActive` event arrived yet;

(iii) For each target process of one of these operations, initiates a ping-pong communication with the place that hosts the trace for the target process (intralayer direction); and

(iv) When a place finished all of its ping-pong communications it injects an event with the `cStateAck` hook (primary direction).

Action (i) ensures that module instances do not advance to subsequent operations, which avoids the creation of a continuous stream of communication events. As the previous example illustrates for point-to-point operations, `passSend` and `recvActive` events can be in transfer between pairs of places on the match layer. Action (ii) identifies all pairs of places for which such outstanding events may exist. Finally, action (iii) initiates a ping-pong communication between each of these place pairs, for which the place that hosts the send operation sends the `ping` event and the place that hosts the receive operation answers with a `pong` event. This communication ensures—due to GTI's order preserving communication—that any outstanding `passSend` event is processed before a respective *ping* event. Thus, a place that hosts a receive operation is guaranteed to inject any outstanding `recvActive` events before it injects the *pong* event. This in turn guarantees that a place that hosts a send operation will process and handle any outstanding `recvActive` events before it handles the *pong* event. Appendix B.5 (page 171) details this ping-pong communication with its analyses and hooks.

After a place completed all its ping-pong communications, it injects an event with the `cStateAck` hook (primary direction) to notify the *WfgManager* module that it arrived in a consistent state (action (iv)). The *WfgManager* uses the `wfgDataRequest` hook to inject an event along the broadcast direction to request the actual wait-for data when it received `cStateAck` events from all match layer

places. The use of the `cStateAck` and `wfgDataRequest` events serves two purposes: First, if places would immediately provide WFG data after the ping-pong communications, then places that only host receive operations would not know when the ping-pong communications are completed. Second, inconsistent states can also result from pending `collActive` and `collAck` events. The `cStateAck` and the subsequent `wfgDataRequest` events ensure that any pending communications on the primary or broadcast directions arrive before a place handles a `wfgDataRequest` event. GTI's order preserving event aggregation system guarantees that `collActive` events precede `cStateAck` events and that `collAck` events precede `wfgDataRequest` events.

5.5.2 WFG Data

When an instance of the *TransitionSystem* module handles a `wfgDataRequest` event, it builds wait-for dependencies for each active and blocking operation to which no transition rule applies. Previous publications [67, 69] describe the wait-for conditions that result from individual operation types. For point-to-point operations, the *TransitionSystem* module retrieves matching information from the *P2PMatch* module in order to build their wait-for dependencies. Similarly for blocking completions, the module combines the wait-for dependencies of unmatched point-to-point operations that are associated with a completion. The type of the completion operation determines the semantic of this combination. If a *TransitionSystem* module instance must report wait-for dependencies for a collective operation, it only describes the communicator of the collective. The *WfgManager* then determines which processes did not join a collective to compute the actual wait-for dependencies.

The *WfgManager* waits until all places of the match layer reported their wait-for dependencies, builds a WFG from this information, and applies a graph search to detect deadlock (if present). If it detects a deadlock, it reports it to the user and terminates its execution. Otherwise, the *WfgManager* module acknowledges the receival of the wait-for information with an event that uses the broadcast direction. When *TransitionSystem* module instances receive this event, they unlock their state and continue their analysis.

5.5.3 Time Complexities

The synchronization scheme requires that each place of the match layer performs a ping-pong communication with all other places in worst case, which requires $\mathcal{O}(p)$ time for p processes. Each communication of the places on the match layer with the *WfgManager* module requires $\mathcal{O}(\log p)$ time, if an aggregation combines the individual replies from the match layer (assuming idle places). An unmatched wildcard receive operation introduces p arcs into a WFG. A completion operation that is associated with k such wildcard receives then introduces $k \cdot p$ arcs and k additional nodes [76]. With that, the overall transfer of wait-for information to the *WfgManager* module requires $\mathcal{O}(k \cdot p^2 \cdot \log p)$ time in worst case, if each process issues a completion with k wildcard receive operations. The actual deadlock detection then requires $\mathcal{O}(k^2 \cdot p^2)$ time in worst case ($k > 0$). Thus, a worst case synchronization, WFG data communication, and deadlock detection requires $\mathcal{O}\big(k \cdot p^2 \cdot (k + \log p)\big)$ time in total.

Since the timeout is configurable and should appear rarely, the $\mathcal{O}(p)$ communication time for the ping-pong communications could have a limited impact on tool scalability. More efficient synchronization schemes could use communication along the primary and broadcast direction instead. However, the p^2 factor for the wait-for data communication and the actual deadlock detection clearly limit scalability. Future extensions of the graph-based detection approach could use more compact representations to transfer wait-for data and could combine WFG nodes for processes with similar behavior. Approaches for stack-based debugging techniques [7] highlight that most execution states in a parallel application will have many commonalities across the individual processes. Approaches such as ScalaExtrap [171] provide techniques to efficiently encode process groups and could highlight that most processes in MPI parallel applications exhibit similar behavior.

Graph-based deadlock detection with the approach in this section may remain an option for applications at smaller scale, but will ultimately limit scalability. At the same time, the transition system of

the previous section already suffices to detect deadlock. Disabling graph-based detection starting at a given scale is consequently an option to overcome this limitation. However, at the cost of disabling the advanced error reports [124] that a graph-based deadlock detection enables.

5.6 Previous Publications

A first publication on MUST [78] specifies the design goals of MUST and primarily details GTI functionality. Further publications detail specific checks [125] or output extensions [124] of MUST. This document focuses on the distributed analyses that enable scalable runtime MPI verification and is orthogonal to these publications.

The description of the point-to-point matching module in this chapter relates to a previous publication [75] with a similar topic. The description in this document introduces the use of GTI in more detail, especially Appendix B.3 (page 169) elaborates on the adaption of the point-to-point matching analysis to runtime decisions of the MPI implementation. A further difference to the previous publication is the time complexity analysis that specifies the analysis cost of MUST's point-to-point matching capability.

The distributed, hierarchical scheme for MPI collective verification that this document introduces is based on three publications: A first prototype to match and check argument consistency for a single MPI collective, a first full prototype for hierarchical collective checking [63], and a publication on the final implementation in MUST [70]. This document uses the module-hook illustration in Section 5.3.3 to highlight how the hierarchical collective matching approach can efficiently utilize GTI. In particular, the first prototype [63] did not fully support MPI calls that require non-transitive type matching checks, such as `MPI_Gatherv`. Additionally, this document provides more detail for the intralayer-based communication for type matching information of collectives with non-transitive rules than the previous description of the MUST implementation [70]. A further difference to the previous documents is the clear definition of the type matching handling with redundancy information for `MPI_Allgatherv` and `MPI_Reduce_scatter`. Finally, Table 5.2 introduces a summary of the *CollectiveMatch* module handling for different classes of collectives. Both previous publications lacked the time complexity analysis that this table includes.

The transition system in Section 5.4 is based on a previous publication [68] and restates its formalization. However, the description of the distributed transition system implementation in Section 5.4.5 then provides more detail on the design of the *TransitionSystem* module and how GTI serves to provide its necessary tool communication. A mayor difference in the implementation that this chapter introduces is a more efficient handling of point-to-point operations. The previous implementation used three communication events to handle a pair of point-to-point operations, whereas the scheme that this document introduces only requires two events. Finally, the time complexity analysis in Section 5.4.8 provides a clear definition of the analysis costs of the distributed transition system implementation, where the previous publication lacked this investigation.

Both the *WfgManager* module in Section 5.5 and a previous publication [68] apply graph-based deadlock detection to states of the distributed transition system implementation. The description in this document, however, provides more detail on how GTI provides the necessary means of communication to both implement synchronization and wait-for data communication. In addition, the time complexity analysis in this document clearly defines the scalability limitations of this approach. Both descriptions rely on the results of previous approaches to handle MPI deadlock with centralized graph-based approaches [67, 69, 76].

5.7 Comparison

This chapter presents the overall design of a prototype runtime verification tool called MUST. The tool provides similar functionality to Umpire [159] and overcomes its corner case limitations with the use of a formal transition system and a well-defined graph-based deadlock detection [67, 69, 76]. Tools such

as Marmot [99], TAC [120], MPI/SX [153], and MPICH-Coll [40] have similar goals, but use timeout-based deadlock handling approaches, no deadlock approach at all, or can provide false-negatives for type matching checks. With that, the approach in this document advances non-scalable implementations of precise checks towards distributed implementations. At the same time it avoids the simplifications that limit the precision of TAC, MPI/SX, and MPICH-Coll, which provided their basis for scalability. The time complexity analysis in this chapter highlights that all analyses except the graph-based deadlock detection can scale for some scenarios. The graph-based deadlock detection itself is not necessary to detect deadlocks, which the transition system provides already, but can provide detailed error reports where application scale permits.

The approach in this chapter chooses to analyze a single execution of an MPI application and uses return values from MPI calls to adapt its matching decisions to the ones of the MPI implementation. Tools such as ISP [156], specifically analyze alternative interleavings to detect deadlock in other executions. MUST avoids this approach, which enables the low time complexities for the deadlock detection approach in this chapter. Exponential search spaces and assumptions on the behavior of MPI applications[3] limit both applicability and scalability of ISP. The approach in DAMPI [162] uses even stronger assumptions on application behavior[4] and the limited precision of Lamport clocks [103] to reveal all alternative interleavings of an MPI application with a single execution. Afterwards, DAMPI uses a timeout-based approach to detect deadlock in all interleavings. The MUST prototype could provide a precise deadlock detection approach for the individual executions that enforce a particular interleaving instead. This combination would overcome the limited precision of DAMPI's timeout approach.

Finally, this chapter details how MUST is implemented as a collection of GTI modules and how it utilizes individual capabilities of GTI to implement its novel analyses. The use of modules simplifies the development of the tool and also provides well-defined means for documentation, such as the module-hook charts that Section 5.2.1 introduces. Automatic utilities around GTI could generate such charts from a tool specification. Further, if modules provide time complexities for their analyses, then future extensions of GTI could also evaluate the scalability of a tool, based on a tool specification and a tool layout. Such an analysis could reveal scalability bottlenecks immediately to highlight tool limitations or design flaws. Particularly, MUST requires capabilities such as event aggregation, intralayer communication, and order preserving event aggregation, which—to the best of my knowledge—no other parallel tools infrastructure provides in combination. This notion underlines the applicability of GTI for a complex and scalable tool. Additionally, the GTI prototype enables flexible configurations of MUST, where the user of the tool can decide whether he is interested in all of its correctness checks or just a subset. As an example, if a particular user experiences an application crash, a run without deadlock detection can reduce tool overheads. Experience with an on-line tracing prototype [89, 165] shows that some of MUST's modules provide a high degree of reusability. The mentioned prototype tool reuses several MUST modules, e.g., call location services from Appendix B.1 (page 167).

[3] Execution of MPI operations must only have non-deterministic control-flow dependencies that depend on return values of MPI operations.

[4] Processes must execute similar series of MPI operations across all executions.

6 Application Study

> *Finally, we noted the debugging value of trying large-scale experiments early and often during the benchmarking process. Too often, we increased the scale of our experiments only to expose bugs in our software or experimental methodology that were hidden at the smaller scales.* [130]

In order to evaluate the applicability and scalability of the prototype implementations of GTI and MUST, this chapter applies synthetic stress tests and benchmark applications. The synthetic tests exhibit distinct types and patterns of MPI operations to enable a step-by-step investigation of the behavior of the proposed correctness analyses. The benchmark applications then serve as more complex test cases that provide feedback on applicability for real world applications. Section 6.1 introduces details on the overall measurement setup. Synthetic tests then specifically stress the individual components from the previous chapter to allow a comparison of exhibited behavior and theoretic time complexities (Section 6.2). A comparison to a centralized implementation within MUST compares the scaling behavior of the distributed components to previous centralized runtime correctness approaches. Further, these measurements evaluate the performance impact of increased numbers of tool places. Section 6.3 then uses the SPEC MPI2007 [116] benchmark suite to apply MUST to strong scaling tests that are derived from real world applications. A first goal of these measurements is to evaluate the applicability of MUST to more complex applications. Secondly, the overhead results for this benchmark provide information on the performance impact that the tool could exhibit for potential target applications. Section 6.4 uses the NAS Parallel Benchmarks (NPB) [10] that consist of synthetic kernels and pseudo applications to evaluate MUST's behavior at increased scale. An overall evaluation of the measurement results concludes this chapter (Section 6.5).

Besides the performance oriented experiments in this chapter, MUST uses a test suite with a wide range of examples—with or without a defect—to evaluate whether its various correctness analyses operate as intended. This includes examples that specifically target the distributed MUST components from the previous chapter (point-to-point matching, collective matching, and deadlock detection). Particularly, this suite includes the test cases from the runtime MPI correctness checking tools Marmot [99] and Umpire [159] to include challenging test cases that previous approaches identified.

6.1 Measurement Setup

The application study in this chapter uses the Sierra cluster at the Lawrence Livermore National Laboratory and the Juqueen system at the research center Jülich. The former system is Linux-based and consists of 1,944 nodes with two 6 core Xeon 5660 processors each. Nodes have 24 GB of available main memory and are connected with QDR InfiniBand. Juqueen uses the BlueGene/Q architecture and features 28,672 nodes, each equipped with 16 cores and 16 GB of main memory. Microarchitectures on the front- and back-end nodes of Juqueen differ. Thus, the system employs a cross compiler.

Sierra supports all features of MUST and allows experiments with up to 8,096 MPI processes. Particularly, this system supports the application crash-handling scheme in MUST and resembles other systems that provided results for the SPEC MPI2007 benchmark. The latter enables a comparison of benchmark runtimes to published results, which helps to identify situations in which an application exhibits unexpectedly slow results. Thus, Sierra serves for both the synthetic stress tests and for experiments with SPEC MPI2007. Juqueen provides far more compute cores than Sierra, while it also represents a second architecture type. Thus, Juqueen serves for experiments with NPB at increased scale, while it also evaluates the applicability of MUST and GTI to different compute architectures.

The measurements in this chapter evaluate the overall performance impact of MUST, which includes its tool stack that consists of GTI and the base infrastructure P^{n}MPI. This holistic approach to determine overall tool-induced overheads provides results on the scalability of the whole prototypes of MUST and GTI. Experiments with different tool configurations then provide information on the behavior of individual correctness analyses, as well as on the impact of the tool layout. Available documentation [66] details the access to the source code that any measurement employs, the installation of the overall tool stack, the setup and installation of the target benchmarks, access to synthetic benchmarks or utilities, and access to result files. This information serves as a reference for an increased level of detail, as well as for providing an opportunity to reproduce the results in this chapter.

Firstly, this section summarizes the tool configurations that the measurements use. Afterwards, an additional description of the GTI prototype implementation details how GTI provides tool places and which communication choices the individual MUST configurations use. Finally, this section provides information on the MPI process-to-core placement that impacts measurement results.

6.1.1 Metric and Layouts

An application run with the MUST prototype increases both the required number of compute resources and the runtime of the target application. This chapter focuses on the impact of the increased runtime. The increase in involved compute resources depends on the tool layout, where subsequent experiments require up to twice as many compute resources. However, this resource cost is bounded for a given tool layout, whereas runtime can drastically increase if a tool fails to scale. Thus, additional compute resources for runs with the tool—during testing and debugging phases of the development workflow—increase cost by up to a factor of two in the following measurements. However, this cost increase is less critical than severe runtime increases, which, for a tool that fails to scale, can increase linearly with scale, or even worse.

As a consequence, this chapter focuses on the runtime increase that results from the presence of the MUST prototype. Thus, the main metric for the experiments in this chapter is the *slowdown factor* with the tool, i.e., a ratio of the runtime with a specific MUST layout to the runtime of a reference run (no tool). For brevity, *slowdown* is subsequently used to refer to the slowdown factor. A slowdown of 1 indicates that the tool causes no (measurable) overhead, while a slowdown of 2 indicates that the tool doubles the execution time.

The measurements analyze three different MUST layouts that specifically evaluate the overheads of the correctness analyses of the previous chapter:

- Layouts that only map the *P2PMatch* module,
- Layouts that only map the *CollectiveMatch* module, and
- Layouts that only map the *TransitionSystem* and *WfgManager* modules, which indirectly use both the *P2PMatch* and the *CollectiveMatch* modules.

The first layout allows a detailed study of the overheads that point-to-point matching and its associated type matching induce. This allows a study of whether layouts with the *P2PMatch* module can achieve the scalability that the theoretic time complexities in Table 5.1 (page 89) suggest. The second layout then specifically targets the overheads of the collective matching and collective consistency checks. Synthetic kernels for different collectives then allow a comparison of actual overheads to the time complexities from Table 5.2 (page 96). Finally, the third layout allows a study of the distributed transition system implementation and its centralized deadlock detection. Again this layout targets a basic comparison of measured overheads to the time complexities of Table 5.3 (page 107).

Besides the modules for the above types of correctness analyses, the individual layouts use any utility modules, e.g., resource tracking or logging modules, that they require. As a result, since the measured overheads include GTI and P^{n}MPI overheads, as well as overheads from multiple utility modules, this

chapter only targets a comparison to the theoretic time complexities, rather than a micro-benchmark approach that fits measurements of individual GTI/MUST components to their theoretic time complexities. The latter would need to use predefined operation schedules, as to consider the processing order and queue length factors in the time complexities for the *P2PMatch*, *CollectiveMatch*, and *TransitionSystem* modules.

6.1.2 Places and Communication

GTI uses communicator virtualization [99] to utilize MPI processes as both application and tool places. This approach allows MUST to instantiate, i.e., start and initialize, all places with a single *mpiexec* command. Thus, batch system requests must allocate compute cores for both the application processes and the tool places of a MUST layout. As an example, a layout for 4 application processes and two tool layers—of 2 and 1 places each—would require a batch allocation of 7 compute cores. MUST would then use an *mpiexec* command that requests 7 processes in total. GTI then virtualizes the communicator `MPI_COMM_WORLD`, such that the application processes only perceive 4 of these processes, while the remaining 3 processes serve as tool places. This technique allows all measurements in this chapter to utilize MPI communication, i.e., all measurements use an MPI-based communication protocol (Section 4.2.4 on page 46). The only exceptions are MUST layouts that use the application crash-handling scheme of GTI, which uses a shared memory communication protocol in addition (Section 4.3.5 on page 60).

GTI provides a flexible selection of a communication strategy in order to enable a consideration of optimization for bandwidth or latency. The MUST layouts that employ the *P2PMatch* and *CollectiveMatch* modules apply a communication strategy that aggregates multiple events into larger continuous communication buffers (100 KiB), in order to optimize for higher bandwidths. The *TransitionSystem* layout, on the other hand, must both tolerate a potential application deadlock and is more latency sensitive, since it requires replies for control messages. Thus, this layout uses a communication strategy that immediately (asynchronous up to a limit of outstanding messages) sends events for latency efficiency. This both removes the need to flush aggregated event buffers on the application processes when a deadlock occurs, and it reduces the latency for control messages of the *TransitionSystem* module. This choice impacts the overhead of the *TransitionSystem* module layout, since it will not be able to utilize the available bandwidth as efficiently as the other two layouts.

6.1.3 Process-to-Core Placement

For each application or benchmark, the experiments in this chapter execute one or multiple configurations of MUST, along with a reference run that does not use the tool. The reference runtime then serves to calculate a slowdown for each MUST configuration. The measurements execute all of these runs (for a single target benchmark) in the same batch job, i.e., with the same compute node allocation. A process-to-core mapping must determine which MPI processes execute on which compute cores. This mapping impacts application performance. Processes that regularly communicate with each other will often benefit from a mapping that puts them onto the same compute node, or at least onto compute nodes that have a short distance in the network topology. Thus, all experiments ensure that the application processes of reference and of tool runs execute on the same compute cores. This increases the comparability between executions with or without the MUST prototype.

With 12 available cores per compute node on Sierra, all measurements map 12 application processes on each compute node, starting with the first compute node that the batch system provides. When all application processes are mapped, the process-to-core mapping continues with the processes for the tool places, also using up to 12 tool places per compute node. The mapping on Juqueen is similar, but uses 16 application processes per compute node, since they provide 16 cores.

As a result, tool places usually execute on different compute nodes than application processes. Process placement optimizations [17, 19, 83] would provide better process-to-core mappings that could consider both the communication of the application itself, as well as the added communication of the MUST prototype. Some of the experiments on Sierra use the application crash-handling scheme of MUST. In

order to enable a shared memory communication, this scheme requires that one tool place executes on each compute node that executes any application processes. Thus, this layout differs from the above placement in that is uses at most 11 application processes per compute node. As to be comparable, reference runs for the crash-handling layouts use 11 application processes per compute node only.

6.2 Synthetic Tests

The first set of measurements in this chapter uses synthetic stress tests to evaluate how the MUST prototype behaves for pre-defined communication patterns. These stress tests include:

- The kernel *cyclicexchange* that only employs point-to-point communication;
- One collective kernel for each class of collectives in Table 5.2, e.g., *reduces* for `MPI_Reduce`; and
- The two deadlock patterns *dlcyclic* and *dlall*.

The point-to-point kernel *cyclicexchange* resembles the *Exchange* pattern that the Intel MPI benchmarks [86] introduce, but differs in its time measurements. It uses the MPI operations `MPI_Isend`, `MPI_Recv`, and `MPI_Waitall`. The collective kernels each use a single collective operation to evaluate how MUST handles them. Finally, the two deadlock patterns target the centralized deadlock detection of the *WfgManager* module to analyze whether its limited scalability yields acceptable overheads at up to 4,096 processes.

Each of these kernels executes its point-to-point pattern or its collective in a loop for a set amount of iterations. This approach resembles a weak scaling test where the workload remains constant or even increases with scale. Increasing workloads particularly apply to several MPI collectives, such as for the kernel *alltoalls* that uses `MPI_Alltoall` operations. As a result, measurements for some of the collective kernels use iteration counts that decrease with scale, as to avoid excessive runtime requirements. The deadlock kernels cause a deadlock in their first iteration. Additionally, all MPI operations use the datatype `MPI_INT` along with a count of 1. For collectives with arrays of counts, the kernels set each array entry to 1. Finally, all kernels use the MPI communicator `MPI_COMM_WORLD` exclusively. The detailed documentation for these measurements [66] contains access information to the implementation of each kernel.

MPI initialization and finalization overheads increase with scale on Sierra, i.e., the cost of executing `MPI_Init` and `MPI_Finalize`. Would the measurements in this section include these increasing overheads for a weak scaling benchmark, then their impact would bias the resulting slowdowns. Thus, the synthetic kernels measure the time that elapses after rank 0 leaves `MPI_Init` and before it enters `MPI_Finalize`. This time span exclusively includes the execution time for the actual communication pattern or collective. MUST's asynchronous operation mode allows tool places to still analyze some events when rank 0 issues `MPI_Finalize`. In order to include processing costs that take place while the application already enters `MPI_Finalize`, runs with MUST calculate the time span on the tool place that serves as root. The root place will issue `MPI_Finalize` only after all event analysis was completed.

This section presents three studies:

- Fan-in measurements for the *cyclicexchange* and the *reduces* (`MPI_Reduce`) kernels to evaluate the performance impact of different application-to-tool-place ratios (fan-ins), this includes reference measurements with centralized reference modules in MUST;
- A study that analyzes the slowdown of the *CollectiveMatch* module for different collective kernels; and
- A study that analyzes the applicability of the *WfgManager* module.

6.2.1 Fan-In Study

Figures 6.1 and 6.2 presents slowdowns for the *cyclicexchange* and *reduces* kernels on Sierra. The *P2PMatch* layout (Figure 6.1(a)) and the *TransitionSystem* layout (Figure 6.1(b)) apply to the *cyclicexchange* kernel. The *CollectiveMatch* layout is not meaningful for this kernel, since the kernel only employs point-to-point communication operations (except for initialization and finalization). Similarly, the *CollectiveMatch* layout (Figure 6.2(a)) and the *TransitionSystem* layout (Figure 6.2(b)) apply to the *reduces* kernel. For this kernel, the *P2PMatch* layout is not meaningful, since this kernel only employs collective operations.

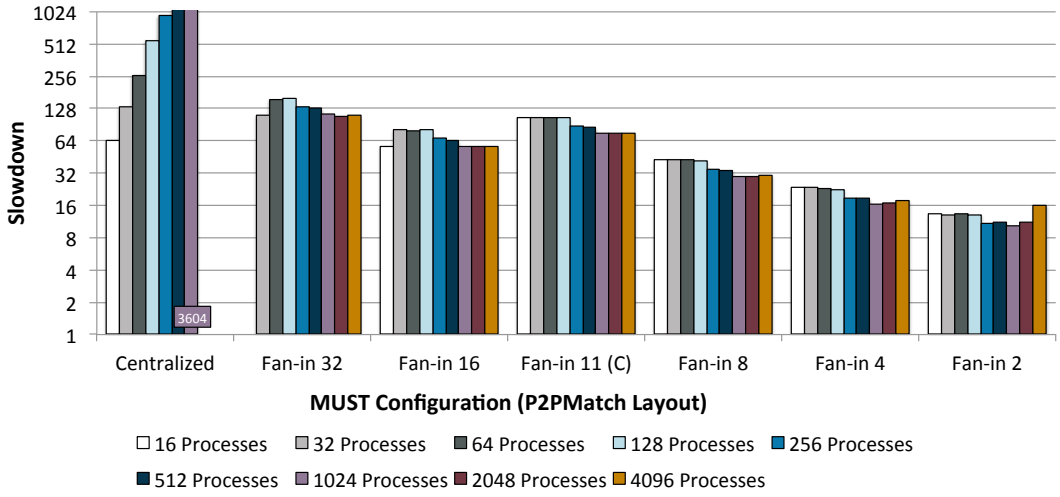
The study uses layouts with fan-ins of 32, 16, 11, 8, 4, and 2. Runs with a fan-in of 11 use the application crash-handling scheme rather than a purely MPI-based communication. In addition, the measurements compare the slowdowns of the MUST layouts to centralized reference implementation that use a single tool place for their tool analysis ("Centralized" in Figures 6.1 and 6.2). Exponential fittings highlight the general behavior of such centralized implementations that fail to scale for these weak scaling experiments.

6.2.1.1 Cyclicexchange

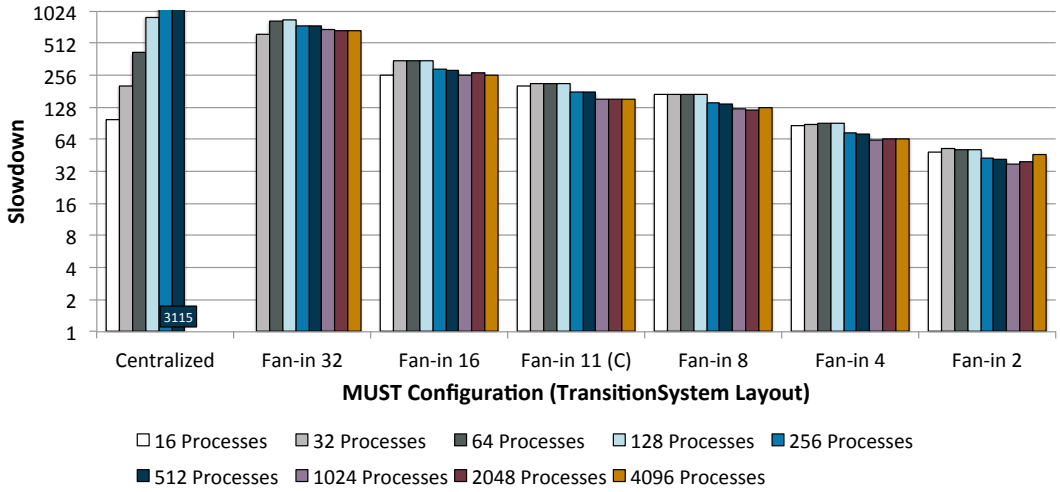
The *cyclicexchange* kernel achieves good weak scaling properties. Its communication wall clock time increases from about 0.2 seconds for 16 processes to about 0.3 seconds at 4,096 processes. Thus, the total number of events that a tool needs to analyze doubles with each increase in scale, while the kernel runtime stays roughly equal for each increase. The centralized implementation fails to cope with this increasing workload, while the layout that only maps the *P2PMatch* module—"fan-in 32" to "fan-in 2" in Figure 6.1(a)—demonstrates that the module implementation can handle this increasing workload without increasing overheads. As expected, higher fan-ins yield higher overheads, while low fan-ins decrease the tool overhead. Overheads for each fan-in stay about constant across scale. A fan-in of 2 achieves a slowdown of 16.2 for 4,096 application processes. The subsequent sections of this chapter relate this stress test result to the behavior of actual applications.

The measurements with a fan-in of 11 in Figure 6.1(a) yield higher slowdowns than a fan-in of 16. This behavior results from the application crash-handling scheme for fan-in 11. This scheme uses shared memory instead of the MPI communication medium between the first tool layer and the application processes. Additionally, the crash-handling layout uses a communication strategy that immediately sends events rather than aggregating multiple events into larger continuous buffers. These differences in the communication choices cause increased communication overheads on the application processes, as well as on the places of the first tool layer. Thus, causing the increased overheads for fan-in 11.

In Figure 6.1(a), for 4,096 processes both fan-in 2 and fan-in 4 have almost identical slowdowns, whereas each other scale clearly exhibits a performance benefit for fan-in 2. An analysis of GTI's profiling data highlights that this behavior results from increased MPI communicator management and setup costs for the experiments with fan-in 2. At 4,096 processes, this communicator setup consumes about 36.0% of the total measured wall clock time, whereas for fan-in 4, it only consumes 6.2% (13 layers with a total of 8,191 processes versus 7 layers with a total of 5,461 processes). This behavior highlights a potential for future study and improvement, but it does not indicate a deficiency in designs of GTI or the *P2PMatch* module. Particularly, the profiling results from GTI detail that the overall average time per analysis function remains about constant across scale and is almost similar for both fan-in 2 and fan-in 4. This behavior matches the theoretic time complexities from Table 5.1 (page 89). The stress test uses a single communicator only, no wildcard receives, and the type matching cost for the basic datatype `MPI_INT` is $\mathcal{O}(1)$ [125]. Thus, the theoretic time complexity to analyze each operation in the *cyclicexchange* kernel is $\mathcal{O}(l)$. The list size l remains, since the kernel specifically uses multiple tags to explore the impact of this factor. The almost constant average time per analysis function across scale (about $1\mu s$) highlights that this factor has close to no impact for the *P2PMatch* implementation with the *cyclicexchange* kernel. At the same time, this experiment underlines that the implementation of both GTI and the *P2PMatch* module can match the theoretic time complexities of Table 5.1. Except for the increasing



(a) Slowdowns for the *P2PMatch* layout highlight about constant slowdowns across scale, while a centralized reference implementation fails to scale.



(b) Slowdown for the *TransitionSystem* layout highlight increased slowdowns that remain about constant across scale, while a centralized reference implementation fails to scale.

Figure 6.1: *Evaluation of the fan-in impact for the* cyclicexchange *kernel on the Sierra system.*

management overheads with lower fan-ins, the performance results support that decreasing fan-ins reduce the tool slowdown. For the Sierra system with a target scale of up to 4,096 processes, a fan-in of 4 provides a good balance between extra resources and the resulting performance improvements.

Figure 6.1(b) presents the slowdowns of the *TransitionSystem* layout for the *cyclicexchange* kernel on Sierra. This layout uses the *TransitionSystem*, *P2PMatch*, *CollectiveMatch*, and the *WfgManager* modules. The latter two modules impact the overall slowdown only marginally. The *CollectiveMatch* module remains almost completely inactive since the kernel uses no collective operations, except for initialization and shutdown. The *WfgManager* module only activates after a timeout, which, according to GTI's profiling data, happens once or twice during most experiment runs. WFG data analysis consumes about 10.0% of the total runtime for fan-in 2 at 4,096 processes on the root place of the layout. The analysis of the *TransitionSystem* and *P2PMatch* modules on the first tool layer dominate the overall tool overhead across all fan-ins and all levels of scale.

Like the *P2PMatch* layout, the *TransitionSystem* layout achieves almost constant slowdowns across scale. At 4,096 processes, a fan-in of 2 yields a slowdown of about 46, which highlights both the

performance impact of the *TransitionSystem* module and the lack of aggregating communication for this layout. Particularly, since all fan-ins cannot use an aggregating communication strategy, the application crash-handling scheme for fan-in 11 causes almost no extra overhead. An analysis of the overall average time per analysis function highlights that analysis time increases with scale. For fan-in 2 it increases from $1.8\mu s$ at 16 processes to $2.1\mu s$ at 4,096 processes. At the same time the maximum trace size for this fan-in increases from 182 operations at 16 processes to 6,058 operations at 4,096 processes. Thus, the trace length l influence—as $\mathcal{O}(\log l)$—in the theoretic time complexity for the *TransitionSystem* module in Table 5.3 (page 107) could cause this increase in the average times. Overall, the scaling behavior of the *TransitionSystem* module in Figure 6.1(b) highlights that the approach in this thesis can yield dramatic applicability improvements for runtime deadlock detection. The state-of-the-art [76] reference implementation "Centralized" yields a slowdown of 3,115 at 512 processes as a comparison, i.e, fan-in 2 at 512 processes decreases the tool slowdown by two orders of magnitude already.

6.2.1.2 Reduces

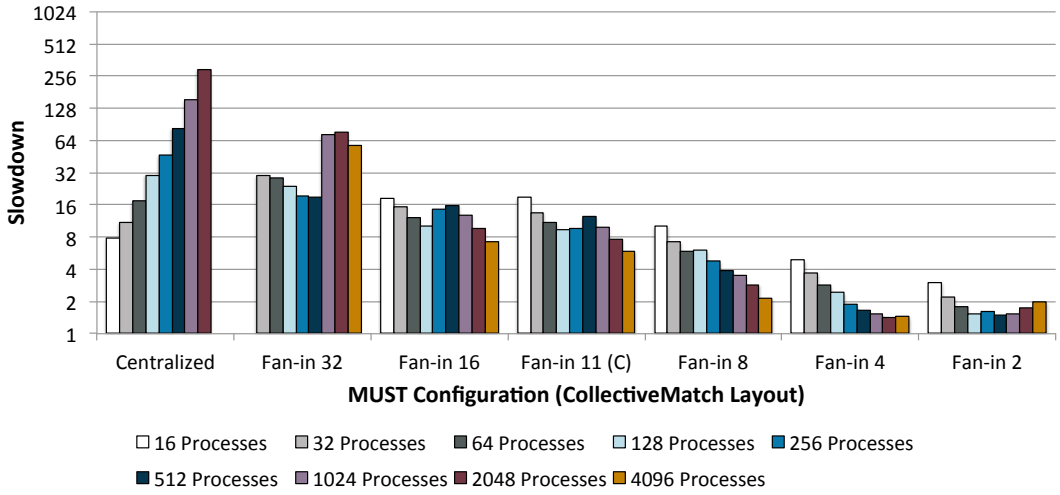
Figure 6.2(a) presents the slowdown of the *CollectiveMatch* layout for the *reduces* kernel on Sierra. This kernel uses the collective operation `MPI_Reduce` that provides weak scaling characteristics. However, the theoretic runtime of the operation must increase at least in a logarithmic manner with scale. With increasing scale, the reference runtimes for this kernel increases. The kernel loop consumes about 0.5s for 16 processes and about 4.3s for 4,096 processes. This increase in reference runtime impacts the slowdown results for the *CollectiveMatch* layout. Slowdown for fan-in 4 starts at 4.9 for 16 processes and decreases towards 1.4 for 4,096 processes. GTI's profiling data highlights that the average analysis function runtime (on the first tool layer) remains about constant across scale. Thus, the about constant event analysis cost across scale, along with increasing reference runtimes, enable decreasing tool overheads in Figure 6.2(a). In other words, MUST adapts to the increase in scale better than the underlying MPI implementation.

Table 5.2 (page 96) summarizes the theoretic time complexities for the *CollectiveMatch* module. The *reduces* kernel uses a single MPI communicator only and causes no timeouts during the experiments, i.e., it aborts no event aggregations. Thus, since no collective waves exist for the timed-out state, the analysis time for each `MPI_Reduce` operation is $\mathcal{O}(1)$ for this kernel. The about constant analysis times from GTI's profiling data suggests that the implementation matches this cost for the kernel.

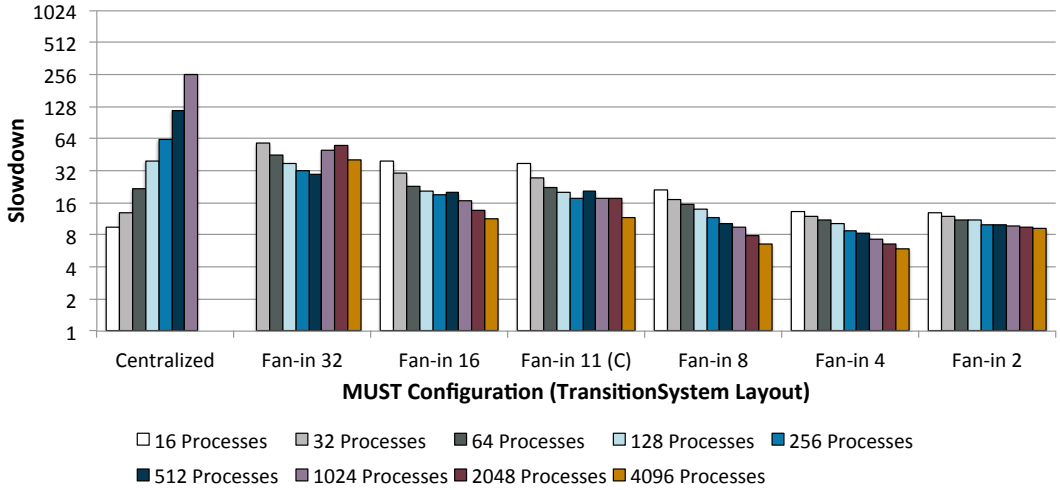
The slowdown for fan-in 32 in Figure 6.2(a) exhibits a strong increase for 1,024 processes. GTI's profiling data indicates that the primary source for this slowdown is the root place of the layout. The data reports a substantial amount of runtime to infrastructure activities such as running the main loop of a place or executing order preserving event aggregation. The source of this overhead remains a target for future study and could point to a performance inefficiency in the source code. This defect seems to introduce an overhead that at least linearly depends on the fan-in of a place, since low fan-ins such as 4 do not exhibit this behavior. In addition for fan-in 32, layouts for 64, 128, 256, and 512 processes use a fan-in of 2, 4, 8, and 16 respectively for their root process (these process counts are no powers of 32). Thus, these levels of scale suffer less from the assumed performance inefficiency.

Figure 6.2(a) highlights that the slowdown for fan-in 2 decreases from 16 processes up to a scale of 1,024 processes. Afterwards, the slowdown increases with scale. With 1,024 processes, the slowdowns for both fan-in 2 and fan-in 4 are about equal. At higher scale, fan-in 2 yields higher slowdowns than a fan-in of 4. GTI's profiling data highlights that MPI communicator setup also impacts the measurements with the *reduces* kernel. For fan-in 2, at 4,096 processes, 25.1% of the measured time is spent for this communicator management. Whereas for fan-in 4 at the same scale, this activity only consumes about 7.3% of the measured time. Thus starting at a scale of 1,024 processes, the increasing communicator setup cost negates the decreased analysis load per tool place that fan-in 2 achieves over fan-in 4.

GTI's profiling data also highlights that MPI activities on the application layer dominate the runtime for fan-in 2 and 4 runs with 1,024 processes or more. Tool places report high idle times for these levels of scale and fan-ins. Figure 6.2(a) reports a slowdown of about 1.5, even though GTI's profiling



(a) The slowdown for the *CollectiveMatch* layout remains about constant or even decreases with scale, while a centralized reference implementation yields slowdowns that exponentially increase with scale.



(b) The slowdown for the *TransitionSystem* layout remains about constant or even decreases with scale, while a centralized reference implementation yields slowdowns that exponentially increase with scale.

Figure 6.2: *Evaluation of the fan-in impact for the* reduces *kernel on the Sierra system.*

data indicates idle times on all tool places. A likely cause for this behavior is that the kernel's MPI communication competes with GTI's MPI-based communication, thus increasing MPI runtimes on the application processes. Overall, a slowdown of 1.5 for 4,096 processes with fan-in 4 is a very encouraging result for a stress test benchmark.

Figure 6.2(b) presents slowdowns for the *TransitionSystem* layout for the *reduces* kernel. The slowdowns exhibit similar anomalies for fan-ins 2 and 32 as for the *CollectiveMatch* layout. Since the *TransitionSystem* layout uses a communication strategy that immediately sends events—rather than aggregating events into larger continuous buffers—it causes higher overheads than the *CollectiveMatch* layout. Additionally, the *TransitionSystem* module introduces additional analysis overheads and uses control messages between the first tool layer and the root of the layout (Section 5.4.5 on page 102). For each `MPI_Reduce` collective, the instances of the *TransitionSystem* module must wait until their control messages propagate to the root, which then provides a reply. The module instances can continue their analysis only after a reply arrives. This distinction impacts the slowdowns in Figure 6.2(b), since this reply time logarithmically increases with the number of application processes (Section 5.4.8 on page 107).

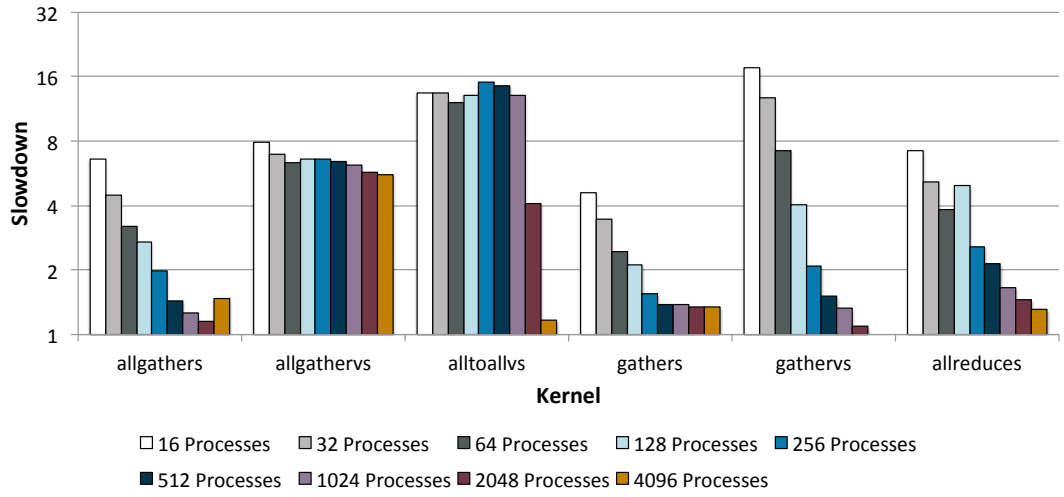


Figure 6.3: *Slowdowns for the* CollectiveMatch *layout on Sierra with fan-in* 4 *highlight that the proposed distributed design can scalably analyze representatives of the collective classes from Table 5.2.*

However, as for the *CollectiveMatch* layout, slowdown still decreases with scale, e.g., for fan-in 4 it decreases from 13.3 at 16 processes to 5.9 at 4,096 processes. The average runtime per analysis function (on the first tool layer) remains about constant across scale, which supports that the module implementation can match the theoretic time complexities from Table 5.3 (page 107). The scaling behavior of the *TransitionSystem* layout along with its low slowdowns at scale underlines the advantages of this distributed transition system approach for MPI deadlock detection. Particularly, at 1024 processes already, the fan-in 4 layout performs about 35 times faster than the previous state-of-the-art runtime deadlock detection approach "Centralized" [76]. Note that the latter approach specifically optimized the handling of collective operations to achieve an $\mathcal{O}(1)$ handling cost per collective operation.

6.2.2 Collective Study

The behavior of the *CollectiveMatch* module depends on the type of collective operation. As Table 5.2 (page 96) summarizes. Collectives with redundant or non-transitive type matching data can require additional consistency checks, or even require intralayer communication on the first tool layer. Measurements in this section evaluate the behavior of the *CollectiveMatch* module for a representative collective from each of the groups in Table 5.2. The measurements exclude the group that the *reduces* kernel represents, since the previous section already analyzed measurement results for this kernel. All measurements use a fan-in of 4, since this selection yielded low overheads in the fan-in study.

Figure 6.3 presents the measured slowdowns and uses the kernels *allgathers* (`MPI_Allgather`), *allgathervs* (`MPI_Allgatherv`), *alltoallvs* (`MPI_Alltoallv`), *gathers* (`MPI_Gather`), *gathervs* (`MPI_Gathervs`), and *allreduces* (`MPI_Allreduce`). The results highlight that the *CollectiveMatch* module scales well for each of the collective classes, as the theoretic time complexities in Table 5.2 suggest.

The implementation of the module yields decreasing slowdowns for the kernels *allgathers*, *gathers*, *gathervs*, and *allreduces*. GTI's profiling data reports about constant average analysis function runtimes (first tool layer) for these four kernels. Thus, since reference runtimes increase with scale, tool slowdown decreases. The *CollectiveMatch* module uses intralayer communication to implement type matching checks for the *gathervs* kernel. First layer tool places—analyzing an `MPI_Gatherv` event from the root process of a collective—initiate this intralayer communication. The kernel advances the root of the collective in a round-robin fashion (as in the Intel MPI benchmarks [86]). Thus, the extra workload to initiate this intralayer communication is distributed equally across the tool places. As a result, a static

root rank could yield higher slowdowns.

The kernels *allgathervs* and *alltoallvs* yield higher slowdowns for the *CollectiveMatch* layout and do not exhibit a steady slowdown decrease with scale. The *allgathervs* kernel triggers the redundancy type matching checks in the *CollectiveMatch* module. For p application processes, these checks introduce an $\mathcal{O}(p)$ time complexity for the analysis of each operation and also require that GTI communicates events of size $\mathcal{O}(p)$. GTI's profiling data highlights this with average analysis function runtimes that increase linearly with scale. This increasing correctness analysis cost yields no increasing slowdowns, since the implementation of this collective operation also involves an $\mathcal{O}(p)$ time complexity on all application processes.

The *CollectiveMatch* module uses intralayer communication on the first tool layer to handle the necessary type matching checks for the *alltoallvs* kernel. This intralayer communication yields an all-to-all communication pattern on the first tool layer. Each `MPI_Alltoallv` collective creates $\mathcal{O}(p^2)$ intralayer messages $((p/4)^2$ for fan-in $4)$. The kernel uses decreasing iteration counts at higher scale, since the reference runtime for the kernel increases with $\mathcal{O}(p^2)$. These reduced iteration counts reduce the number of intralayer messages that each pair of first tool layer places exchanges with each other. The intralayer communication strategy that the *CollectiveMatch* layout uses provides a capacity limit for outstanding messages. This capacity is available for each target place. With the decreased iteration counts, for $2{,}048$ and $4{,}096$ processes, the intralayer communication strategy does not exceed this capacity anymore. This causes lower intralayer communication overheads for these levels of scale and enables the reduced tool slowdowns in Figure 6.3. This behavior highlights a potential for performance improvements in the intralayer communication strategy in use. However, this deficiency does not limit the scalability of the approach.

Appendix C.1 (page 173) presents slowdown results for the *TransitionSystem* layout with the individual collective kernels. These measurements mainly suffer the performance impact from the immediate communication (rather than aggregating multiple events into a larger continuous buffer). The analyses of the *TransitionSystem* module are independent of the collective operation type, and thus, only impact collective operations with low analysis costs.

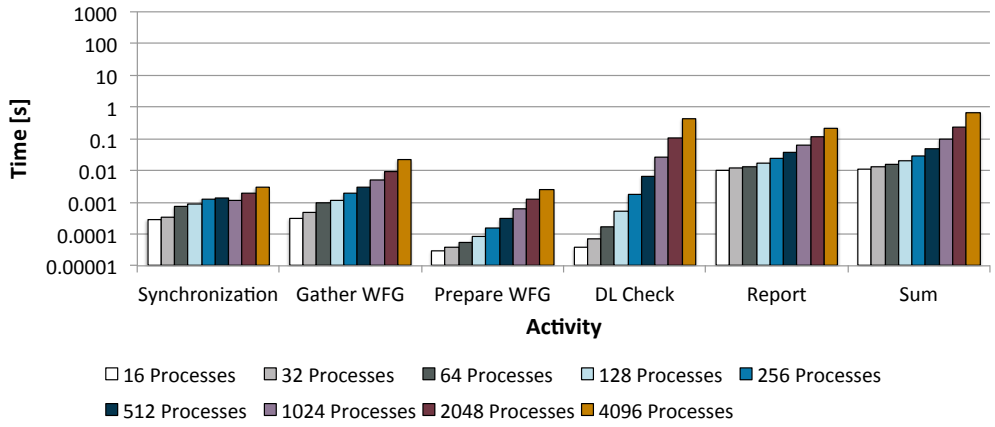
In summary, in all measurements with the synthetic collective kernels, the MUST layouts are able to scale well.

6.2.3 Deadlock Study

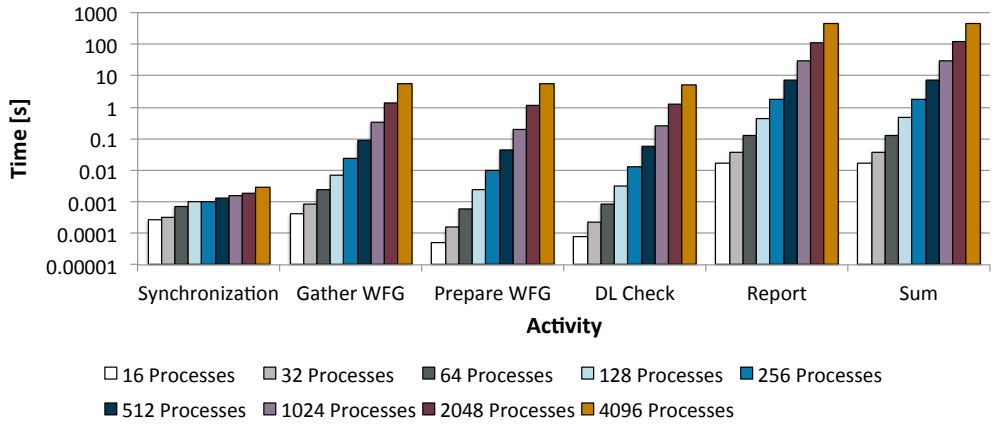
The previous measurements evaluated the overheads of the *P2PMatch*, *CollectiveMatch*, and *TransitionSystem* module implementations. Two deadlocking kernels serve as synthetic tests to evaluate the behavior of the *WfgManager* module. The theoretic time complexities in Section 5.5.3 clearly highlight the limited scalability of this module. As a result, these measurements primarily evaluate whether the approach is still applicable at a scale of up to $4{,}096$ processes. In addition, they analyze the cost of different activities during deadlock detection, including the synchronization time of the *TransitionSystem* module. These individual activities are:

- **Synchronization:** Synchronization time to compute a consistent state in the *TransitionSystem* module (Section 5.5.1);
- **Gather WFG:** Runtime spent to receive wait-for information for all processes on the root of the layout;
- **Build WFG:** Runtime spent to build the wait-for graph on the root of the layout;
- **Deadlock Check:** Runtime of the graph search for a necessary and sufficient deadlock criterion; and
- **Report:** Runtime spent to provide an HTML report, and a wait-for graph of the deadlocked processes in the DOT[1] format.

[1] http://www.graphviz.org/



(a) The breakdown for the *dlcyclic* kernel highlights that WFGs with few arcs impose no scalability limitations with 4,096 processes.



(b) The breakdown for the *dlall* kernel highlights that complex WFGs impose noticeable overheads at 4,096 processes.

Figure 6.4: *A breakdown of the overheads for Wait-For Graph (WFG) analysis with increasing scale on Sierra.*

This section uses the two deadlocking kernels *dlcyclic* and *dlall* that differ in the wait-for dependencies that they use. The former kernel creates a dependency chain with p arcs, whereas the latter uses wildcard receives to create a WFG with p^2 arcs. Figure 6.4 presents a breakdown of the overall deadlock detection cost (after a timeout triggers a detection) for the *dlcyclic* and *dlall* kernels. Besides times for the above activities, it includes a sum for the overall handling as "sum". In Figure 6.4(a), the overall deadlock detection time for the *dlcyclic* kernel increases from 10.8ms for 16 processes to 668.8ms for 4,096 processes. In Figure 6.4(b), the p^2 WFG size of the *dlall* kernel causes higher deadlock detection overheads, which increase from 17.3ms for 16 processes to 470.8s for 4,096 processes. Writing the WFG in the DOT format into a file consumes the majority of the latter runtime (454.7s). Without writing this output, a detection overhead of 16.1s remains for this scale. This overhead is noticeable, but still imposes no restriction on the use of the centralized graph-based deadlock detection with 4,096 processes.

At the same time, a WFG with 4,096 nodes and $4{,}096^2$ arcs provides no meaningful output to tool users. This notion highlights that even if the implementation in MUST was using distributed algorithms to scalably build the WFG and to detect the deadlock criterion, the user output could still not be visualized. As a result, techniques to detect processes that are in a similar execution state could drastically reduce the number of nodes and arcs in a WFG. Techniques such as in ScalaExtrap [171] could facilitate such a regularity detection, while work on stack trace debugging [7] motivates the assumption that most

parallel application states will have a limited (low) number of distinct activity types across all processes. A similarity-based approach to condense wait-for conditions of processes in similar activities would both reduce detection and WFG analysis overheads. At the same time, it would yield outputs that could be visualized for the end user.

Overall, the overheads in Figure 6.4 highlight that a deadlock detection at 4,096 processes remains feasible with 668.8ms overall detection time for the *dlcyclic* kernel and 16.1s detection time for the *dlall* kernel (without the report time). In addition, the overhead breakdown underlines that all of the activities except for the *Synchronization* activity suffer from limited scalability. The latter activity exhibits a sub-linear increase with scale, but also depends on the state of the *TransitionSystem* module, i.e., the number of outstanding control messages that exist when a request for a consistent state arrives.

6.3 SPEC MPI2007

The benchmark suite SPEC MPI2007 [116] (v2.0) provides benchmarks that are derived from real world applications. This includes complex applications such as the whether prediction code *147.l2wrf2* [145]. Thus, the following refers to these kernels as *applications* instead. The maximum size dataset of SPEC MPI2007 is the *lref* data set that supports up to 2,048 processes[2] and includes twelve applications. Appendix C.2 (page 174) provides a comparison of reference runtimes for these applications to reported benchmark results.

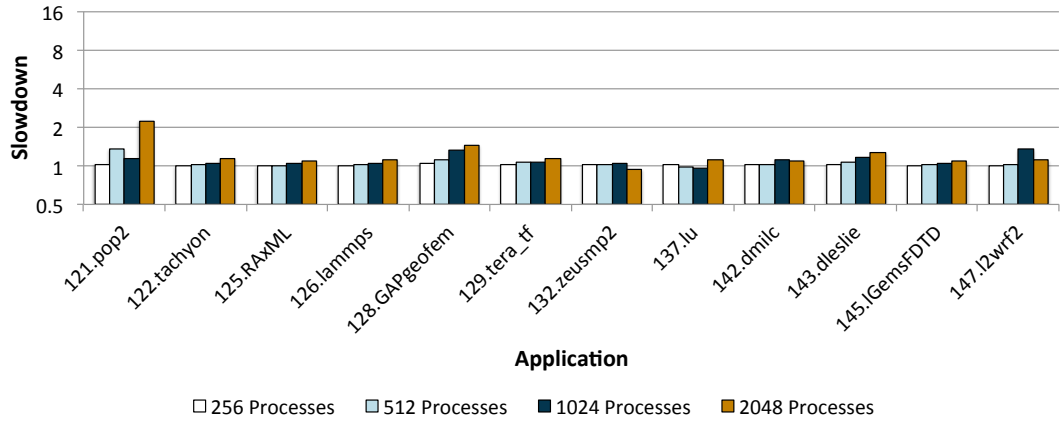
Figure 6.5 presents slowdowns of three MUST layouts with the *lref* data set for SPEC MPI2007. The slowdown calculation uses the benchmark times that the benchmark suite reports, which includes overheads for MPI initialization and finalization. The measurements use a fan-in of 4, which indicated low overheads in the previous measurements. Appendix C.3 (page 175) presents additional results with a fan-in of 11 and GTI's application crash-handling scheme, which is important for use cases that involve an application crash. Figure 6.5(a) presents slowdowns with the *P2PMatch* layout, Figure 6.5(b) uses the *CollectiveMatch* layout, and Figure 6.5(c) uses the *TransitionSystem* layout instead.

Slowdowns for both the *P2PMatch* and the *CollectiveMatch* layouts remain low across scale. All slowdowns for the *P2PMatch* and the *CollectiveMatch* layouts remain below 2, with the only exception of the *P2PMatch* layout for the application *121.pop2* at 2,048 processes. This result highlights that while slowdowns for stress test like the *cyclicexchange* kernel can be high, actual applications exhibit lower loads for a tool such as MUST. SPEC MPI2007 is a strong scaling benchmark where the problem size remains constant and increasing process counts serve to reduce the runtime of an application. Thus, the rate at which application processes issue MPI operations increases with scale. The applications *121.pop2* and *128.GAPgeofem* exhibit the highest rates. At 256 processes, application ranks of *121.pop2* issue up to 3,733 MPI operations per second and at 2,048 processes the application issues up to 21,421 MPI operations per second. Processes of the application *128.GAPgeofem* issues up to 15,170 MPI operations per second at 256 processes and up to 93,411 MPI operations per second at 2,048 processes. This behavior explains the increasing slowdowns in Figure 6.5. This increased slowdown for *121.pop2* at 2,048 processes with the *P2PMatch* layout could be an outlier, since GTI's profiling data reports high idle times on all tool places.

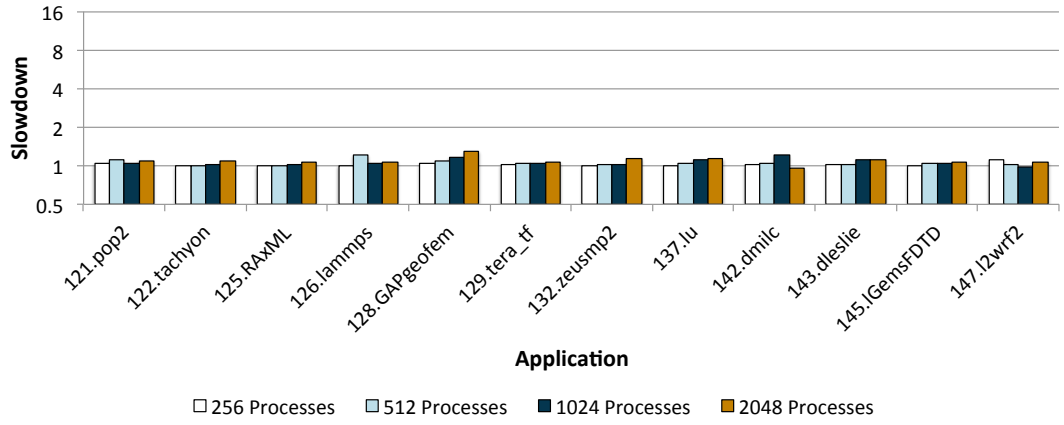
The slowdowns for the *TransitionSystem* layout—Figure 6.5(c)—are higher than for the other two layouts. These increased overheads results both from the lack of an aggregating communication strategy and the additional overheads from the *TransitionSystem* module. The application *128.GAPgeofem* causes slowdowns of up to 15.7. However, this increase is not a scalability limit of the module implementations. The rate at which application processes issue MPI operations increases by a factor of 6.2 from 256 processes to 2,048 processes. The slowdown increases by an almost identical factor of 6.8 as a result.

The application *137.lu* exhibits performance gains—slowdowns below 1—for 256 and 512 processes with the *TransitionSystem* layout. A previous investigation [68] of this anomaly identified high numbers of outstanding `MPI_Send` operations in the application as the source of this behavior. MPI implemen-

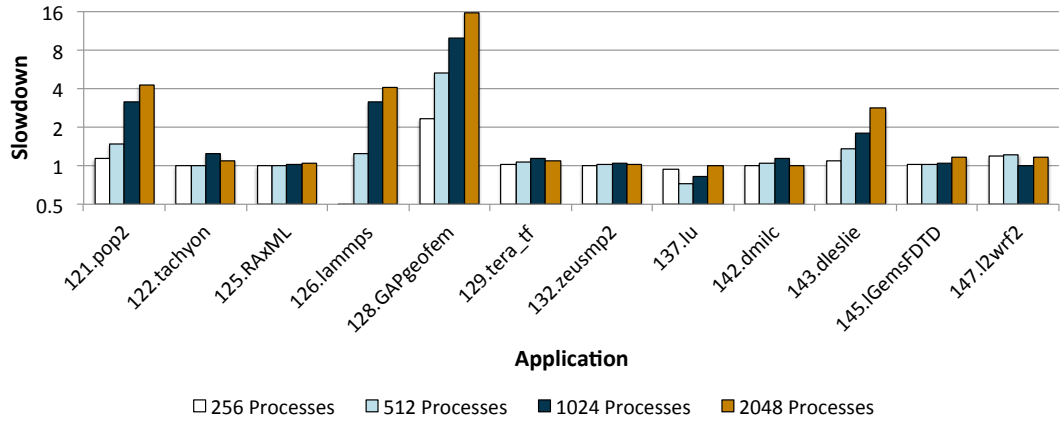
[2] http://www.spec.org/mpi/docs/faq.html#DataSetL



(a) The *P2PMatch* layout imposes low overheads across all applications and all levels of scale.



(b) The *CollectiveMatch* layout imposes low overheads across all applications and all levels of scale.



(c) The *TransitionSystem* layout imposes noticeable overheads for applications with high MPI invocation rates.

Figure 6.5: *Slowdowns for the SPEC MPI2007 applications on Sierra provide information on tool overheads with potential real world applications (*lref *data set and strong scaling).*

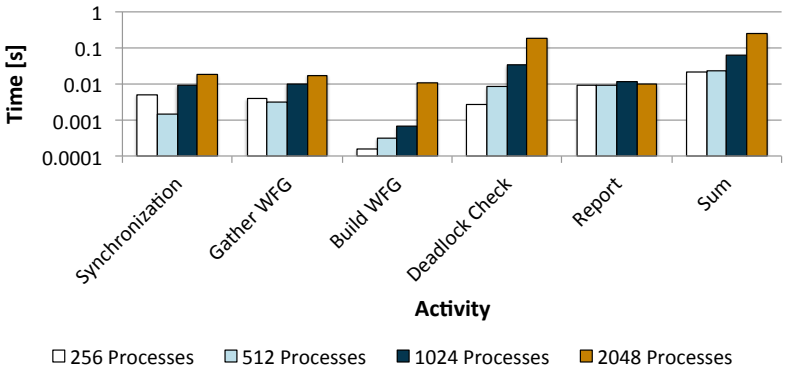


Figure 6.6: *Involved overheads to detect the send-send deadlock in* 126.lammps.

tations usually buffer such operations in order to allow application processes to continue their execution. In order to do so, an MPI implementation must store information for such pending send operations. Whenever an application issues an MPI operation, the MPI implementation checks the progress of all outstanding sends. Large numbers of pending sends then incur a performance penalty whenever the application issues an MPI operation. The communication strategy for the *TransitionSystem* layout blocks within tool internal communication when the first tool layer fails to drain outstanding communication buffers for previous events. This blocking in the tool communication is assumed to reduce the number of pending send operations for *137.lu*. This in turn reduces MPI internal overheads[3]. The previous study used an `MPI_Send` wrapper that rewrites some sends with an `MPI_Ssend` operation to support the soundness of this assumption. This wrapper enabled performance improvements that compare to the runtime reduction for the *TransitionSystem* layout.

The application *126.lammps* contains a send-send deadlock that manifests for MPI implementations that do not buffer the involved `MPI_Send` operations. No deadlock manifests on Sierra, since the MPI implementation on Sierra buffers these operations. At the same time, the application remains incorrect and has limited portability as a consequence. The slowdown for *126.lammps* in Figure 6.5(c) includes the overall runtime of the application with the *TransitionSystem* layout. This time is dominated by MPI internal timeouts that implement the `MPI_Abort` operation on Sierra. The *WfgManager* module invokes this operation after it detects and reports a deadlock. Figure 6.6 provides a breakdown of the deadlock detection overheads for this application with the same activity types as in Section 6.2.3. The overall detection time remains at about 0.24s at 2,048 processes, since the deadlock criterion involves a pair of processes only. Thus, the report visualizes a deadlock criterion of constant size across scale.

Overall, the *P2PMatch* and the *CollectiveMatch* layouts exhibit low slowdowns for all of the SPEC MPI2007 applications and enable a correctness analysis for benchmarks that are based on actual applications. Slowdown for the *TransitionSystem* module is below 2 for eight out of twelve applications, while the layout detects a defect for the application *126.lammps*. Slowdown for the remaining three applications is in a range of 2.8 up to 15.7. The former slowdown may not cause issues for some use cases, but could limit applicability for long running applications. However, this effect is not a cause of limited scalability, but rather a result of high handling costs with GTI and the MUST module implementations. Future (single core) performance optimization in these implementations could reduce the overheads further.

[3]Note that GTI itself suffered from similar performance penalties when it used high numbers of pending send operations for tool communication. Performance tuning revealed that a limit of at most 50 outstanding send operations yields dramatic performance improvements.

6.4 NAS Parallel Benchmarks

This section evaluates module behavior at increased scale on the Juqueen system. The NAS Parallel Benchmarks (NPB) [10] (v3.3) serve as target benchmark on Juqueen and support a scale of up to 16,384 processes. This benchmark is also a strong scaling benchmark. The experiments use problem size *D* with up to 4,096 processes and size *E* for increased scale (up to 16,384 processes). The experiments increase the process count from 512 to 16,384 processes, which requires the use of two problem sizes, as to avoid excessive runtimes and out-of-memory situations at smaller scale.

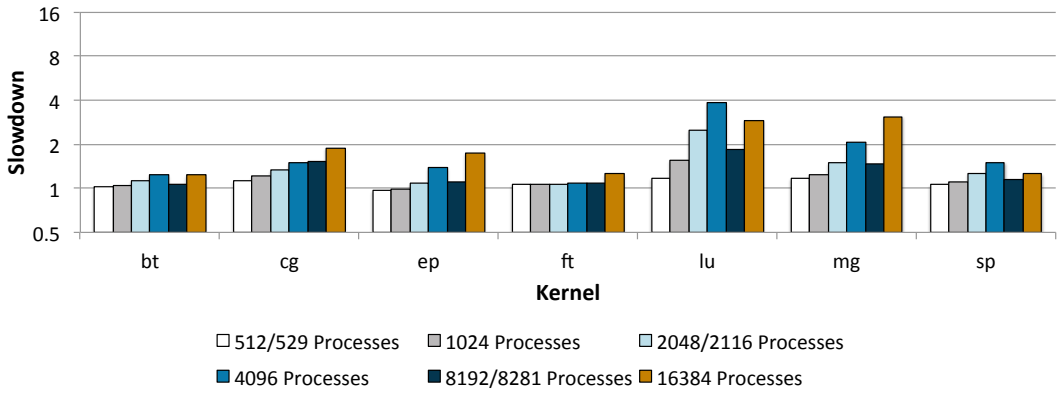
All runs use 16 MPI processes per compute node. As opposed to the measurements for SPEC MPI2007 that use a fan-in of 4, initial measurements with this fan-in on Juqueen yielded moderate to high slowdowns. Thus, the experiments in this section use a fan-in of 2 between the application layer and the first tool layer. This choice provides additional parallelism to the layer that hosts the most modules for the *TransitionSystem* layout. All remaining layers use a fan-in of 4, in order to limit the amount of extra resources that the runs require. Benchmark results exclude the kernel *is*, as it does not support problem size *E*. The kernels *bt* and *sp* require numbers of processes that are a square of a natural number. As a result, these kernels use 529 instead of 512, 2,116 instead of 2,048, and 8,281 instead of 8,192 processes.

Figure 6.7 presents slowdowns for the kernels and pseudo applications from NPB. The measurements use the *P2PMatch* layout (Figure 6.7(a)), the *CollectiveMatch* layout (Figure 6.7(b)), and the *TransitionSystem* layout (Figure 6.7(c)). An initial slowdown calculation from total execution times—as for SPEC MPI2007—highlighted moderate to high slowdowns, especially for short running kernels such as *mg* and *ep*. Reference measurements with an application that only invokes `MPI_Init` and `MPI_Finalize` highlight that without the tool, runtime increases from 4.6s for 512 processes to 5.5s for 16,384 processes. At the same time, total runtime with the MUST layouts increases from 20.5s for 512 processes to 82.1s for 16,384 processes (numbers for the *P2PMatch* layout). Thus, the MUST modules invoke a noticeable and scale dependent startup/shutdown overhead. A preliminary investigation detected increasing overheads for loading shared libraries as one influential overhead. A future study of the individual startup and shutdown overheads could reduce the overheads that MUST introduces. Particularly, an approach such as Spindle [47] could reduce shared library loading times. At the same time, a constant overhead of about 80s at 16,384 processes (including writing of profiling data) is noticeable, but should not limit the applicability of the approach. As a consequence, the slowdown calculation in Figure 6.7 uses the reported benchmark times from NPB as the reference runtime, which captures a timespan that starts after the application issued `MPI_Init` and that ends before it invokes `MPI_Finalize`. The MUST runtime captures the same timespan, but removes startup and shutdown overheads that include initialization times for loading shared libraries and for MPI communicator management.

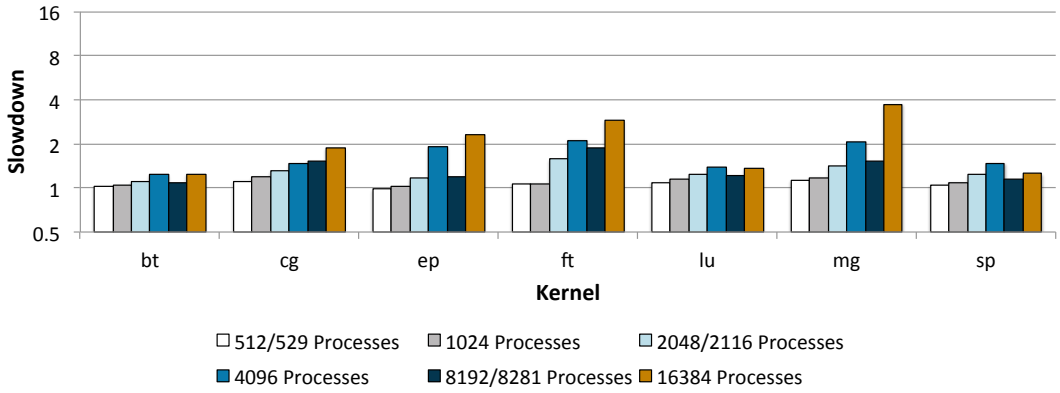
Overall, Figure 6.7(a) presents low slowdowns for the *P2PMatch* layout. Only the kernels *lu* and *mg* exhibit slowdowns above 2. The former kernel issues large numbers of point-to-point messages that stress the *P2PMatch* module. The dip in the slowdown at 8,192 processes results from switching from problem size *D* to problem size *E*. This switch increases runtimes for the kernel and causes reduced MPI call invocation rates. These reduced invocation rates enable lower slowdowns for MUST.

The kernel *mg* has a very short reference runtime—5.5s for 16,384 processes—and issues MPI operations at a low rate. The increasing slowdowns for this kernel result from an increased load on the root place of the MUST layouts. An investigation of this overhead highlights that initialization events for MUST's MPI resource tracking modules (utilities) increase with scale. This behavior represents a target for future improvements, but does not highlight a limitation in the distributed analysis modules that this thesis proposes.

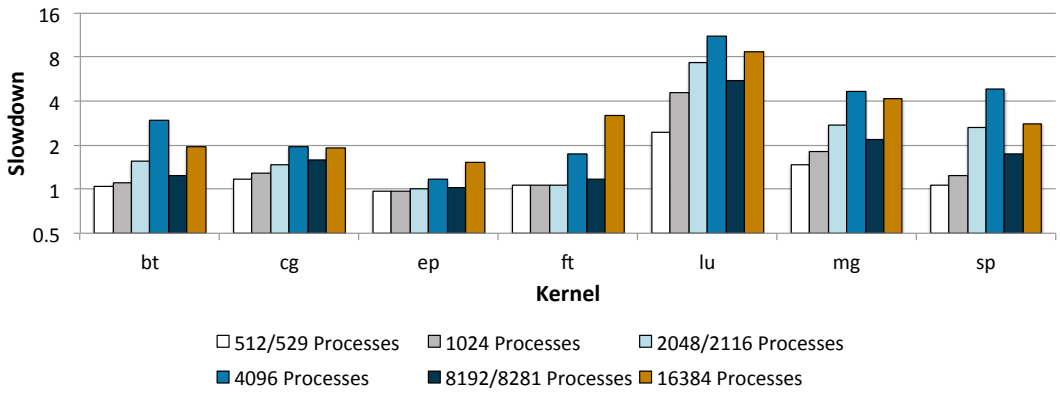
The slowdowns for the *CollectiveMatch* layout (Figure 6.7(b)) are generally low. The kernels *ep*, *ft*, and *mg* exhibit slowdowns above 2. Since *lu* primarily uses point-to-point operations, it stresses the *CollectiveMatch* layout far less than the *P2PMatch* layout. The *mg* kernel suffers from the same effect as for the *P2PMatch* layout. The slowdown for the kernel *ep* is slightly above 2 for 16,384 processes and combines workload for collectives and a short runtime (7.4s for 16,384 processes). As a result, it exhibits a slightly higher load for the *CollectiveMatch* layout than for the *P2PMatch* layout, while overheads for



(a) Slowdowns for the *P2PMatch* layout depend on MPI invocation ratios and remain below 4.



(b) Slowdowns for the *CollectiveMatch* layout depend on MPI invocation ratios and remain below 4.



(c) The *TransitionSystem* module imposes increased overheads for kernels with high MPI invocation ratios.

Figure 6.7: *Slowdowns for kernels and pseudo applications of NPB on Juqueen provide results for the behavior of the MUST prototype at increased scale (problem size* D *up to* 4,096 *processes and size* E *for higher levels of scale).*

MPI resource tracking initialization again cause an increased overhead at scale. The kernel *ft* stresses the *CollectiveMatch* module since it uses collectives on small user-defined communicators. These small and comb-shaped communicators (non-continuous ranks) reduce the efficiency of the event aggregations that the module employs (Section 4.5.5 on page 77).

Figure 6.7(c) presents NPB slowdowns for the *TransitionSystem* layout, which largely combines the overheads of the previous two layouts. Particularly, slowdowns for *cg*, *ep*, *ft*, and *mg* closely resemble the slowdowns of the *CollectiveMatch* layout. Again the lack of aggregating communication strategies and the additional overheads of the *TransitionSystem* modules increase slowdowns for the remaining three point-to-point intensive kernels *bt*, *lu*, and *sp*.

6.5 Summary

This chapter evaluates overheads of the prototype implementations of GTI and MUST. Synthetic stress tests highlight that tool layouts with the *P2PMatch*, *CollectiveMatch*, and *TransitionSystem* modules scale very well. Depending on the kernel and the fan-in, all layouts exhibit near constant or even decreasing slowdowns across scale. These empirical results support the theoretic time complexities of the previous chapter and highlight that the runtime correctness approach that this thesis proposes can scale with applications. Comparisons with centralized implementations for deadlock detection highlight performance improvements of two orders of magnitude for 512 processes already.

An evaluation of the *WfgManager* module, which provides limited scalability, reveals its applicability to complex deadlock scenarios for up to 4,096 processes. Experiments with these deadlock examples highlight that solutions that remove the scalability limitations of this module must not only distribute the actual analysis, but must also provide condensed outputs to the user. These results motivate the exploration of similarity-based reduction techniques for wait-for data.

The stress tests can cause the prototype implementation to exhibit high slowdowns even for low fan-ins. Measurements with two benchmark suites relate these results to more complex synthetic kernels, pseudo applications, and benchmarks derived from real world applications. The individual layouts yield low slowdowns—below 2—for most of these 19 examples. Additionally, MUST successfully identifies a potential deadlock in one of the SPEC MPI applications. On Juqueen, the experiments use a scale of up to 16,384 application process, for which the MUST layouts use 27,307 MPI processes in total (fan-in 2 combined with fan-in 4).

Finally, these measurements do not only evaluate the modules from the previous chapter, but also the GTI prototype as a whole. Most importantly, the measurements demonstrate that a mapping-based abstraction, as in GTI, not only enables the creation of tools, but that such tools can operate for a wide range of applications across multiple compute architectures. MUST as a tool advances the state-of-the art in runtime deadlock detection for MPI, while it operates on a flexible and generic tool infrastructure at the same time. Thus, this empirical tool study highlights that the abstractions in GTI allow the specification of complex tools that can achieve good scalability. The measurements also underline the applicability of the new infrastructure techniques of this thesis, including mapping-based tool instantiation, intralayer communication, and order preserving event aggregation.

7 Conclusions and Future Work

> *When knowledge is embedded in a tool, it frees the practitioner from the need to master that particular knowledge, it changes the skills needed, and it opens the way for development of new knowledge.* [157]

Parallel programming with message passing abstractions such as MPI introduces an additional source for program defects. The parallelism that enables complex state-of-the art simulations complicates the detection and removal of these defects. Tools that aid application developers and system support personnel in spotting, tracking, and removing such defects add to efficient development workflows. One such class of tools are runtime correctness checkers for MPI applications.

7.1 Conclusions

This thesis advances runtime correctness tools for MPI applications towards increased scalability without sacrificing detection precision. Previous approaches achieved scalability by replacing precise detection algorithms with heuristic techniques that can yield false positives or false negatives (limited precision). Tools that used precise techniques, on the other hand, used centralized error detection algorithms that limited their scalability. The thesis contributes MUST as a novel runtime correctness tool that:

- Uses event offloading like previous runtime correctness tools with deadlock detection capabilities, but uses hierarchies of additional processes rather than a single process to run the correctness analysis;
- Provides distributed precise type matching checks for MPI point-to-point operations;
- Provides hierarchical correctness analyses for MPI collective operations that include precise type matching checks;
- Specifies a transition system that serves for deadlock detection with a state that consists of a single vector;
- Implements a distributed transition system for deadlock detection—based on the small state—that employs control messages within the tool processes;
- Provides detailed deadlock reports with a wait-for dependency analysis that employs a graph-based deadlock criterion; and
- Details the time complexities of these analyses to evaluate their scalability on a theoretic level and to simplify comparisons for future approaches.

Thus, the design and the prototype implementation of MUST fill the gap of a precise tool that offers a high degree of scalability. The theoretic time complexities of the described analyses—except for graph-based reports—highlight that the tool can analyze MPI operations with costs that compare to their overheads within an MPI implementation.

At the same time, the development of runtime tools for high performance computing systems is expensive and involves many common components and tasks. Recent developments in parallel tool infrastructures provide one approach to avoid reoccurring development costs for such tools. Moreover, abstractions and patterns can guide tool developers towards scalable implementations. Several scalable tools use tree-overlay abstractions, such as provided by the abstractions of MRNet. However, this abstraction does not

incorporate instrumentation of a target application and thus requires the use of additional tool components. The runtime correctness use case also requires further functionality classes such as application crash-handling. High event rates for correctness tools also motivate flexible communication systems that can benefit from specialized hardware of modern HPC platforms. Thus, this thesis contributes a concept for a novel tool infrastructure that:

- Incorporates tree-overlay networks with an instrumentation system;
- Provides flexible choices for both the communication medium and the communication timing;
- Reuses the idea of event-action mappings to allow tool developers to specify which tool analyses apply to which events;
- Extends this mapping approach to tree-overlay networks with a notion of event aggregations;
- Allows tools to inject events along three communication directions;
- Overcomes deficits of pure tree-overlay networks for tasks such as point-to-point matching with a new intralayer communication direction;
- Adds a crash-handling scheme to be fully applicable for the runtime correctness use case; and
- Scalably preserves event order in the presence of event aggregations.

The GTI abstraction targets a tool development that allows the programmer to focus on his tool's analyses (functionality) rather than on implementing common components. Most importantly, the tool infrastructure abstraction that this thesis proposes uses a single concept to apply tool analyses onto application processes, the root of a hierarchy of tool places, or on an intermediate hierarchy level. This ability forms a key difference to existing infrastructure concepts, which use differing interfaces with application processes, a root process of a hierarchy, or an intermediate layer. The analysis concept of the GTI abstraction allows tool developers to flexibly apply all types of tool activities, regardless of previous placement decisions. This result is an important contribution towards a development with highly reusable tool components. Additionally, this thesis provides a concept for tool infrastructures that automatically preserve process local event order in the presence of event aggregations. Thus, a tool developer can simply assume that events arrive in their original order, as opposed to using separate event streams or managing event order manually. This further simplifies the development of runtime tools with the proposed tool infrastructure abstraction.

MUST serves as a challenging test case for GTI, which requires a wide variety of its functionality classes. GTI both supports all of MUST's requirements and allows this tool to be specified as a collection of modules—representing the tool analyses—and specifications that provide the analysis-event mappings. Besides that, MUST does not implement any of the usual tool components, e.g., instrumentation, tool internal communication, or spawning extra tool processes/threads.

Widespread use of an abstraction such as GTI would drastically simplify tool integrations and combinations. The integrations of many existing runtime tools for HPC consider pairs of tools and can easily fail after a few releases, due to lack of maintenance. In addition, the whole specification of GTI's abstraction imposes close to no assumptions on the target programming paradigm. While the MUST and GTI prototypes implement their functionality for MPI, GTI could support other paradigms as well. Similarly, GTI can support multiple types of tools. MUST represents a runtime correctness tool, while other GTI-based tool prototypes target a performance optimization use case already. Widespread use of a tool infrastructure such as GTI could provide support for a multitude of programming paradigms along with well-tested scalable communication services. In such a situation, tool developers for GTI automatically gain the capability to easily extend the scalability of their tool and to support further paradigms. These two tasks consume large amounts of development costs in current runtime tool developments, due to an ongoing investigation of new programming paradigms in the HPC community along with increasing system scale.

An empiric study with synthetic stress tests and two widely used HPC benchmarks evaluates the prototypes of GTI and MUST. The stress tests with up to 4,096 application processes (8,191 total MPI processes) underline that the analyses in MUST and the overheads within GTI can match the scalability that their theoretic time complexities suggest. In addition, a study with deadlock cases highlights that the limited scalability of the graph-based deadlock reports causes increased detection times at scale, but remains practical at up to 4,096 processes. The two benchmark suites at 2,048 and 16,384 application processes respectively (2,731 and 27,307 total MPI processes) highlight that the MUST analyses—without deadlock detection—provide low overheads that should not limit the applicability of the approach. With the transition system for deadlock detection, the measurements exhibit more noticeable overheads for the benchmark suites and increase application runtime by a factor of 4 for two out of nineteen kernels. Two further applications exhibit even higher slowdowns at their maximum scale. However, the experiments with the synthetic kernels highlight that this behavior is not a scalability limitation, but rather the result of MPI invocation rates that increase with scale. Future performance optimizations of the MUST and GTI prototypes could reduce these overheads.

At the same time, no other runtime deadlock detection approach for MPI reports overheads for a similar scale. A comparison of the overheads of the distributed transition system to a state-of-the-art centralized runtime deadlock detection approach (also based on GTI) highlights that the approach in this thesis provides two orders of magnitudes lower overheads at 512 processes already. Finally, the measurements use two compute architectures to demonstrate that GTI's abstraction is applicable for multiple architectures, including a frontend-backend system.

7.2 Future Work

Both GTI and MUST serve as prototypes to explore new ideas, designs, and abstractions. These prototypes undergo nightly tests with hundreds of test cases and regular performance tests. As a result, both codes reached a quality that goes beyond a mere prototype and were included in several tutorials and workshops. Furthermore, MUST is installed on HPC systems at the Lawrence Livermore National Laboratories (USA), the Jülich Supercomputing Center (Germany), the Barcelona Supercomputing Center (Spain), the RWTH Aachen university (Germany), and the Technische Universität Dresden (Germany). Several ongoing, planned, and future evaluations and extensions target an evolution of these prototypes towards actual products. Beyond ongoing improvements in applicability, portability, and stability, there exists a wide range of ideas for future research.

7.2.1 Towards Products

The distributed transition system for deadlock detection currently lacks two extensions to support all MPI usage scenarios. The first is an extension to adapt the currently static specifications of which MPI operations may block. Support for *unexpected* matches (Section 5.4.4 on page 101) requires information on whether the MPI implementation buffers a send operation or not. An MPI interface that provides an interface for such information would simplify support for unexpected matches. Secondly, the centralized reference implementation for deadlock detection [76] supports situations where information on wildcard receive matching is unavailable. This implementation uses a state exploration that can yield exponential numbers of states to visit. The distributed implementation in this thesis avoids the use of this technique, due to potentially high overheads. The MPI function `MPI_Request_get_status` (MPI-2 and above) could serve as a solution to avoid explorations of exponential numbers of states.

Placement drivers must select a communication channel to receive new events in their main loop (`choose` function in Section 4.3.6 on page 61). This selection impacts the behavior of a GTI tool if it uses event aggregations or control messages such as for the *TransitionSystem* module. Some newly received events can increase the memory demand of the tool—e.g., a new operation in the trace of the *TransitionSystem* module—while other events can reduce the memory demand. A tool can exhaust the available main memory if a placement driver repeatedly receives events that increase the memory

footprint. This happens for the application *128.GAPgeofem* for example. The GTI prototype uses a mechanism to pause the execution of the application to avoid such out-of-memory situations. A current study [77] extends this technique of the GTI prototype and explores the impact of static and dynamic communication channel selection approaches.

The MUST modules of the *condition* package (Section 5.1 on page 81) highlight a requirement for a further dependency type in GTI. The current module dependencies of GTI enforce that the infrastructure maps dependent modules onto layers that execute the depending module. The condition modules require a new dependency type that causes the dependent modules to be mapped onto the application layer or a specifically marked layer. Currently, MUST manages this dependency type manually.

Finally, GTI-based tools consist of a set of modules, specifications, and potentially additional helper scripts or applications that invoke the tool. The current prototypes lack functionality to easily package up and to provide a GTI tool to its users. A future packaging concept is an important requirement for tool combinations that consist of multiple GTI-based tools. One approach is a repository concept where a GTI utility would allow users to add repository locations for existing tools. GTI could then automatically install and configure these tools in order to create a single collection of tool modules and their specifications.

Simplifications and tool support for the creation and management of GTI specifications are also an important target for improvement. Currently, module interfaces and some of the specifications contain redundant information. A source code analysis approach could create a subset of the specifications automatically in the future. Additionally, tools to analyze and apply analysis-hook mappings could simplify the mapping of analyses to their target hooks. Such extensions would simplify the development of GTI-based tools.

7.2.2 Runtime Correctness Research

New features of MPI, such as the nonblocking collectives in MPI-3, motivate extensions of MUST. The latter type of collectives, as an example, would primarily impact the *TransitionSystem* module, which would need to support completion operations with associated nonblocking collectives. Besides wider MPI support, the *WfgManagerModule* remains a target for future research. Its limited scalability and the results in this thesis motivate an exploration of similarity based reduction techniques (Section 5.5.3 on page 110) to reduce the overall number of nodes in a WFG. A hierarchical reduction of the WFG data could implement this. A reduction could provide users both accessible deadlock reports and limit the overheads of the graph creation and deadlock criterion search.

Tools such as ISP and DAMPI use a model checking like exploration of alternative schedules to reveal defects that only manifest as an inconsitent state or a failure for some interleavings. MUST explicitly avoids such explorations and targets increased scalability instead. DAMPI uses the limited precision of Lamport clocks along with assumptions on the MPI usage of an application to implement a scalable detection of alternative interleavings. Afterwards, it uses a timeout-based heuristic to report the presence of a deadlock for each interleaving. A combination of DAMPI with MUST could combine an interleaving exploration with precise deadlock detection and suggests a high degree of synergy between the approaches. Ongoing research already combines DAMPI's predecessor tool ISP with MUST. This combination uses ISP to explore different interleavings and MUST to report deadlocks. Also, tool combinations with static analysis approaches such as in the temporal order analysis in STAT [2] could allow MUST to provide additional detail for deadlock situations and other defects it detects.

GTI support for further programming paradigms would allow MUST to apply correctness checks to them. This could include message passing libraries such as MCAPI or libraries or language extensions for partitioned global address space programming, e.g., GASPI, Co-array Fortran, and UPC. Ongoing research projects extend GTI to support the OpenMP paradigm and target correctness analyses for the accelerator concepts of OpenMP 4.0.

7.2.3 Tool Infrastructure Research

Currently the GTI prototype relies on an MPI communicator virtualization technique and an MPI-based communication implementation to spawn and connect its tool places. GTI support for further programming paradigms or libraries can require additional technologies to handle spawning and communication tasks. An ongoing investigation for OpenMP support considers the use of additional threads as tool places.

The order preserving event aggregation in GTI uses channel identifiers to store which processes submitted information to an aggregate event. This scheme works well if all processes, or at least continuous ranges of processes, provide events for an aggregation. MPI communicators of a comb-like shape—e.g., odd processes—stress this technique such as in the kernel *ft* of the NAS parallel benchmarks. A stride representation to mark regular subsets of a continuous process range could overcome this limitation. The GTI prototype uses extra bits in the channel identifiers to store such a stride representation to reduce the overheads that it exhibits for the kernel *ft*. A study of potential representations for non-continuous process sets and the resulting extensions to the tree queue algorithm (Section 4.5.3 on page 4.5.3) is still pending.

GTI uses tool places—processes or threads—to provide additional compute resources to tools. An application run with a GTI-based tool then includes both communication operations from the application and the tool internal communication. An optimization of the process-to-compute-core mapping is important for an efficient use of the overall communication resources. Such a placement must consider the communication pattern of the application and that of the GTI tool. A balance between optimizing for the one or the other communication pattern will yield improved latency and bandwidth at higher scales. Currently GTI uses static placements of processes to compute nodes that could result in a high average communication distance at scale. Future studies should evaluate placement optimization frameworks and heuristics that could yield better placements.

Bibliography

[1] Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Overcoming Scalability Challenges for Tool Daemon Launching. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP '08)*, pages 578–585, 2008.

[2] Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. Scalable Temporal Order Analysis for Large Scale Debugging. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, New York, NY, USA, 2009. ACM.

[3] Alexander Aiken and David Gay. Barrier Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM.

[4] Sadaf R. Alam, Richard F. Barrett, Michael Bast, Mark R. Fahey, Jeffrey A. Kuehn, Collin McCurdy, James Rogers, Philip C. Roth, Ramanan Sankaran, Jeffrey S. Vetter, Patrick H. Worley, and Weikuan Yu. Early evaluation of IBM BlueGene/P. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 23:1–23:12, Piscataway, NJ, USA, 2008. IEEE Press.

[5] Allinea Software. Allinea DDT. http://www.allinea.com/products/ddt/. Last visited on 09/12/2014.

[6] Dorian C. Arnold. *Reliable, Scalable Tree-Based Overlay Networks*. PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 2008.

[7] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack Trace Analysis for Large Scale Debugging. In *Proceedings of the 2010 IEEE 21th International Parallel and Distributed Processing Symposium*, IPDPS '07, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[8] Dorian C. Arnold and Barton P. Miller. Scalable Failure Recovery for High-Performance Data Aggregation. In *Proceedings of the 2010 IEEE 24th International Parallel and Distributed Processing Symposium*, IPDPS '10, Red Hook, NY, USA, 2010. Curran Associates.

[9] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing Parallel Programs with Pin. *IEEE Computer – Computer*, 43(3):34–41, 2010.

[10] David H. Bailey, Leonardo Dagum, Eric Barszcz, and Horst D. Simon. NAS Parallel Benchmark Results. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society.

[11] Bartosz Baliś, Marian Bubak, Marcin Radecki, Tomasz Szepieniec, and Roland Wismüller. Application Monitoring in CrossGrid and Other Grid Projects. In Marios D. Dikaiakos, editor, *Grid Computing*, volume 3165 of *Lecture Notes in Computer Science*, pages 212–219. Springer Berlin Heidelberg, 2004.

[12] Susanne M. Balle, Bevin R. Brett, Chih-Ping Chen, and David LaFrance-Linden. A New Approach to Parallel Debugger Architecture. In Juha Fagerholm, Juha Haataja, Jari Järvinen, Mikko

Lyly, Peter Råback, and Ville Savolainen, editors, *Applied Parallel Computing*, volume 2367 of *Lecture Notes in Computer Science*, pages 139–149. Springer Berlin Heidelberg, 2006.

[13] Suman Banerjee, Christopher Kommareddy, Koushik Kar, Bobby Bhattacharjee, and Samir Khuller. Construction of an Efficient Overlay Multicast Infrastructure for Real-Time Applications. In *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications*, volume 2 of *INFOCOM 2003*, pages 1521–1531, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[14] Valmir C. Barbosa and Mario R.F. Benevides. A Graph-Theoretic Characterization of AND-OR Deadlocks. Technical report, Federal University of Rio de Janeiro, 1998. http://www.cos.ufrj.br/~valmir/es47298.pdf. Last visited on 09/12/2014.

[15] Victor R. Basili, Danelia Cruzes, Jeffrey C. Carver, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Marvin V. Zelkowitz, and Forrest Shull. Understanding the High-Performance-Computing Community: A Software Engineer's Perspective. *IEEE Software*, 25(4):29–36, 2008.

[16] Jean-Baptiste Besnard, Marc Pérache, and William Jalby. Event Streaming for Online Performance Measurements Reduction. In *Proceedings of the 2013 42nd International Conference on Parallel Processing*, ICPP '13, pages 985–994, Washington, DC, USA, 2013. IEEE Computer Society.

[17] Abhinav Bhatele. *Automating Topology Aware Mapping for Supercomputers*. PhD thesis, University of Illinois, Department of Computer Science, 2010.

[18] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 41:1–41:12, New York, NY, USA, 2013. ACM.

[19] Shahid H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30(3):207–214, 1981.

[20] Greg Bronevetsky. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Holger Brunst, Dieter Kranzlmüller, and Wolfgang E. Nagel. Tools for Scalable Parallel Program Analysis–Vampir NG and DeWiz. In Zoltàn Juhàsz, Péter Kacsuk, and Dieter Kranzlmüller, editors, *Distributed and Parallel Systems*, volume 777 of *The International Series in Engineering and Computer Science*, pages 93–102. Springer US, 2005.

[22] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[23] Mark Bull. Unified European Applications Benchmark Suite. Deliverable D7.4 from PRACE Second Implementation Phase Project. http://www.prace-ri.eu/IMG/pdf/d7.4_3ip.pdf, 2013. Last visited on 09/12/2014.

[24] Darius Buntinas, George Bosilca, Richard L. Graham, Geoffroy Vallée, and Gregory R. Watson. A Scalable Tools Communications Infrastructure. In *Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, HPCS '08, pages 33–39, Washington, DC, USA, 2008. IEEE Computer Society.

[25] Darius Buntinas and Gregory R. Watson. STCI—Scalable Tools Communication Infrastructure: Fault Manager. http://stci.wikidot.com/fault-manager. Last visited on 09/12/2014.

[26] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *29th International Conference on Software Engineering*, ICSE 2007, pages 550–559, 2007.

[27] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[28] Zhezhe Chen, Qi Gao, Wenbin Zhang, and Feng Qin. FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, 2010. IEEE Computer Society.

[29] Zhezhe Chen, Xinyu Li, Jau-Yuan Chen, Hua Zhong, and Feng Qin. SyncChecker: Detecting Synchronization Errors between MPI Applications and Libraries. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 342–353, Washington, DC, USA, 2012. IEEE Computer Society.

[30] Doreen Y. Cheng, Jeffrey T. Deutsch, and Robert W. Dutton. 'Defensive programming' in the Rapid Development of a Parallel Scientific Program. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):665–669, 1990.

[31] Lawrence Chung and Julio Cesar Prado Leite. Conceptual modeling: Foundations and applications. chapter on Non-Functional Requirements in Software Engineering, pages 363–379. Springer Berlin Heidelberg, 2009.

[32] Ajanta de Sarkar and Nandini Mukherjee. A Study on Performance Analysis Tools for Applications Running on Large Distributed Systems. *International Journal of Information and Computing Science (IJICS)*, 9:52–67, 2006.

[33] Bronis R. de Supinski, Jeffrey K. Hollingworth, Shirley Moore, and Patrick H. Worley. Results of the PERI Survey of SciDAC Applications. *Journal of Physics: Conference Series*, 78(1):012027, 2007.

[34] Jayant DeSouza, Bob Kuhn, and Bronis R. de Supinski. Automated, Scalable Debugging of MPI Programs with Intel Message Checker. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM.

[35] Robert Dietrich, Thomas Ilsche, and Guido Juckeland. Non-intrusive Performance Analysis of Parallel Hardware Accelerated Applications on Hybrid Architectures. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 135–143, Washington, DC, USA, 2010. IEEE Computer Society.

[36] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. Enabling MPI Interoperability Through Flexible Communication Endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 13–18, New York, NY, USA, 2013. ACM.

[37] Jun Doi. Peta-scale Lattice Quantum Chromodynamics on a Blue Gene/Q Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 45:1–45:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[38] Mohamed Elwakil and Zijiang Yang. Debugging Support Tool for MCAPI Applications. In *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '10, pages 20–25, New York, NY, USA, 2010. ACM.

[39] David. A. Evensky, Ann. C. Gentile, L. Jean. Camp, and Robert. C. Armstrong. Lilith: Scalable Execution of User Code for Distributed Computing. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, HPDC '97, Washington, DC, USA, 1997. IEEE Computer Society.

[40] Chris Falzone, Anthony Chan, Ewing Lusk, and William Gropp. Collective Error Detection for MPI Collective Operations. In Beniamino Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 138–147. Springer Berlin Heidelberg, 2005.

[41] Christopher Falzone, Anthony Chan, Ewing L. Lusk, and William Gropp. A Portable Method for Finding User Errors in the Usage of MPI Collective Operations. *International Journal of High Performance Computing Applications*, 21(2):155–165, 2007.

[42] Shiqing Fan, Rainer Keller, and Michael Resch. Advanced Memory Checking Frameworks for MPI Parallel Applications in Open MPI. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 63–78. Springer Berlin Heidelberg, 2012.

[43] Shiqing Fan, Rainer Keller, and Michael M. Resch. Enhanced Memory debugging of MPI-parallel Applications in Open MPI. In Michael M. Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 49–60. Springer Berlin Heidelberg, 2008.

[44] Colin. J. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[45] Leslie R. Foulds. *Graph Theory Applications*. Universitext. Springer New York, 1992.

[46] Daniel P. Freedman and Gerald M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*. Dorset House Publishing Co., Inc., New York, NY, USA, 3rd edition, 2000.

[47] Wolfgang Frings, Dong H. Ahn, Matthew LeGendre, Todd Gamblin, Bronis R. de Supinski, and Felix Wolf. Massively Parallel Loading. In *Proceedings of the 27th International Conference on Supercomputing*, ICS '13, pages 389–398, New York, NY, USA, 2013. ACM.

[48] George W. Furnas and Jeff Zacks. Multitrees: Enriching and Reusing Hierarchical Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, pages 330–336, New York, NY, USA, 1994. ACM.

[49] James Galarowicz. Project Final Report: Building a Community Infrastructure for Scalable On-Line Performance Analysis Tools around Open|SpeedShop. Technical report, Krell Institute, 2014.

[50] Jim Galarowicz, Don Maghrak, and Bill Hachfeld. Building a Community Infrastructure for Scalable On-Line Performance Analysis Tools also known as Component Based Tool Framework "CBTF". http://ft.ornl.gov/doku/_media/cbtfw/cbtf_madison_2011.pdf. Last visited on 18/08/2014.

[51] Qi Gao, Feng Qin, and Dhabaleswar K. Panda. DMTracker: Finding Bugs in Large-scale Parallel Programs by Detecting Anomaly in Data Movements. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 15:1–15:12, New York, NY, USA, 2007. ACM.

[52] Markus Geimer, Sameer S. Shende, Allen D. Malony, and Felix Wolf. A Generic and Configurable Source-Code Instrumentation Component. In Gabrielle Allen, Jarosław Nabrzyski, Edward Seidel, Geert Dick van Albada, Jack Dongarra, and Peter M.A. Sloot, editors, *Computational Science—ICCS 2009*, volume 5545 of *Lecture Notes in Computer Science*, pages 696–705. Springer Berlin Heidelberg, 2009.

[53] Markus Geimer, Felix Wolf, Brian J.N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[54] Ann C. Gentile, David A. Evensky, and Robert C. Armstrong. Lilith: A Software Framework for the Rapid Development of Scalable Tools for Distributed Computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, HPDC '98, Washington, DC, USA, 1998. IEEE Computer Society.

[55] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. Periscope: Advanced Techniques for Performance Analysis. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[56] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996.

[57] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. Formal Analysis of MPI-Based Parallel Programs. *Communications of the ACM*, 54(12):82–91, 2011.

[58] William D. Gropp. Runtime Checking of Datatype Signatures in MPI. In Jack Dongarra, Péter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1908 of *Lecture Notes in Computer Science*, pages 160–167. Springer Berlin Heidelberg, 2000.

[59] Jonathan L. Gross and Yellen Jay. *Handbook of Graph Theory*. Discrete Mathematics and its Applications. CRC Press LLC, 2004.

[60] John Gustafson. Computational Verifiability and Feasibility of the ASCI Program. *IEEE Computational Science & Engineering*, 5(1):36–45, 1998.

[61] Salman Habib, Vitali Morozov, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Tom Peterka, Joe Insley, David Daniel, Patricia Fasel, Nicholas Frontiere, and Zarija Lukić. The Universe at Extreme Scale: Multi-Petaflop Sky Simulation on the BG/Q. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 4:1–4:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[62] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[63] Fabian Hänsel. Scalable Matching and Correctness Checking of MPI Collectives. Großer Beleg, Technische Universität Dresden, 2012.

[64] Waqar Haque. Concurrent Deadlock Detection in Parallel Programs. *International Journal on Computers and Applications*, 28(1):19–25, 2006.

[65] Les Hatton. The T Experiments: Errors in Scientific Software. *Computational Science Engineering, IEEE*, 4(2):27–38, 1997.

[66] Tobias Hilbrich. Measurement Documentation. https://fusionforge.zih.tu-dresden.de/plugins/mediawiki/wiki/hilbrich-result. Last visited on 09/12/2014.

[67] Tobias Hilbrich. Centralized Deadlock Detection for MPI Applications: Complexity and Parallelization. Diplomarbeit, Technische Universität Dresden, 2008.

[68] Tobias Hilbrich, Bronis R. de Supinski, Wolfgang E. Nagel, Joachim Protze, Christel Baier, and Matthias S. Müller. Distributed Wait State Tracking for Runtime MPI Deadlock Detection. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 16:1–16:12, New York, NY, USA, 2013. ACM.

[69] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. A Graph Based Approach for MPI Deadlock Detection. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 296–305, New York, NY, USA, 2009. ACM.

[70] Tobias Hilbrich, Fabian Hänsel, Martin Schulz, Bronis R. de Supinski, Matthias S. Müller, Wolfgang E. Nagel, and Joachim Protze. Runtime MPI Collective Checking with Tree-Based Overlay Networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 129–134, New York, NY, USA, 2013. ACM.

[71] Tobias Hilbrich, Matthias Jurenz, Hartmut Mix, Holger Brunst, Andreas Knüpfer, Matthias S. Müller, and Wolfgang E. Nagel. An Interface for Integrated MPI Correctness Checking. In Barbara Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, Frans Peters, and Thierry Priol, editors, *Parallel Programming Tools for Multi-core Architectures in conjunction with International Conference on Parallel Computing (ParCo 2009)*, volume 19 of *Advances in Parallel Computing*, pages 693–700. IOS Press, 2010.

[72] Tobias Hilbrich, Matthias S. Müller, Bronis R. de Supinski, Martin Schulz, and Wolfgang E. Nagel. GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 1364–1375, Washington, DC, USA, 2012. IEEE Computer Society.

[73] Tobias Hilbrich, Matthias S. Müller, and Bettina Krammer. MPI Correctness Checking for OpenMP/MPI Applications. *International Journal of Parallel Programming*, 37(3):277–291, 2009.

[74] Tobias Hilbrich, Matthias S. Müller, Martin Schulz, and Bronis R. de Supinski. Order Preserving Event Aggregation in TBONs. In Yiannis Cotronis, Anthony Danalis, Dimitrios Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 19–28. Springer Berlin Heidelberg, 2011.

[75] Tobias Hilbrich, Joachim Protze, Bronis R. de Supinski, Martin Schulz, Matthias S. Müller, and Wolfgang E. Nagel. Intralayer Communication for Tree-Based Overlay Networks. In *42nd International Conference on Parallel Processing (ICPP)*, Fourth International Workshop on Parallel Software Tools and Tool Infrastructures, pages 995–1003, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[76] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 30:1–30:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[77] Tobias Hilbrich, Joachim Protze, Michael Wagner, Matthias S. Müller, Martin Schulz, Bronis R. de Supinski, and Wolfgang E. Nagel. Memory Usage Optimizations for Online Event Analysis. In *Accepted at EASC2014: Solving Software Challenges for Exascale*. Springer Berlin Heidelberg, 2014.

[78] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 53–66. Springer Berlin Heidelberg, 2010.

[79] Lorin Hochstein and Victor R. Basili. The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development. *Computer*, 41(3):50–58, 2008.

[80] Lorin Hochstein, Forrest Shull, and Lynn B. Reid. The role of MPI in development time: A case study. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 34:1–34:10, Piscataway, NJ, USA, 2008. IEEE Press.

[81] Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[82] Watts S. Humphrey. Bugs or Defects? http://www.uqac.ca/flemieux/PRO102/watts-mar99.pdf, 1999. Last visited on 09/12/2014.

[83] Joshua Hursey, Jeffrey M. Squyres, and Terry Dontje. Locality-Aware Parallel Process Mapping for Multi-Core HPC Systems. In *IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 527–531, 2011.

[84] Thomas Ilsche, Joseph Schuchart, Jason Cope, Dries Kimpe, Terry Jones, Andreas Knüpfer, Kamil Iskra, Robert Ross, Wolfgang E. Nagel, and Stephen Poole. Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 49–60, New York, NY, USA, 2012. ACM.

[85] Darrel C. Ince, Leslie Hatton, and John Graham-Cumming. The case for open computer programs. *Nature*, 482:485–488, 2012.

[86] Intel. Intel MPI Benchmarks. http://software.intel.com/en-us/articles/intel-mpi-benchmarks. Last visited on 09/12/2014.

[87] Sreekaanth S. Isloor and T. Anthony Marsland. The Deadlock Problem: An Overview. *Computer*, 13(9):58–78, 1980.

[88] Emily R. Jacobson, Michael J. Brim, and Barton P. Miller. A Lightweight Library for Building Scalable Tools. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 419–429. Springer Berlin Heidelberg, 2012.

[89] Sven Jansky. Performance-Analyse von hoch-parallelen Programmen auf der Basis von Online-Daten. Diplomarbeit, Technische Universität Dresden, 2013.

[90] Tor E. Jeremiassen and Susan J. Eggers. Static Analysis of Barrier Synchronization in Explicitly Parallel Programs. In *Conference on Parallel Architectures and Compilation Techniques*, PACT '94, pages 171–180, Amsterdam, The Netherlands, 1994. North-Holland.

[91] Chao Jin, David Abramson, Minh Ngoc Dinh, Andrew Gontarek, Robert Moench, and Luiz DeRose. A Scalable Parallel Debugging Library with Pluggable Communication Protocols. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12, pages 252–259, Washington, DC, USA, 2012. IEEE Computer Society.

[92] Peter Johnsen, Mark Straka, Melvyn Shapiro, Alan Norton, and Thomas Galarneau. Petascale WRF Simulation of Hurricane Sandy: Deployment of NCSAs Cray XE6 Blue Waters. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 63:1–63:7, New York, NY, USA, 2013. ACM.

[93] Tu Hong Jun and Gregory R. Watson. Scalable Communication Infrastructure. http://wiki.eclipse.org/PTP/designs/SCI. Last visited on 09/12/2014.

[94] Rainer Keller, Shiqing Fan, and Michael M. Resch. Memory Debugging of MPI-Parallel Applications in Open MPI. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September 2007*, volume 15 of *Advances in Parallel Computing*, pages 517–523. IOS Press, 2008.

[95] Richard Kendall, Jeffrey C. Carver, David Fisher, Dale Henderson, Andrew Mark, Douglass Post, Clifford E. Rhoades, and Susan Squires. Development of a Weather Forecasting Code: A Case Study. *IEEE Software*, 25(4):59–65, 2008.

[96] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI Analysis and Checking Tool. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, and Wolfgang V. Walter, editors, *Parallel Computing Software Technology, Algorithms, Architectures and Applications*, volume 13 of *Advances in Parallel Computing*, pages 493–500. North-Holland, 2004.

[97] Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and Matthias S. Müller. MPI Correctness Checking with Marmot. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 61–78. Springer Berlin Heidelberg, 2008.

[98] Bettina Krammer, Valentin Himmler, and David Lecomber. Coupling DDT and Marmot for Debugging of MPI Applications. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007*, volume 15 of *Advances in Parallel Computing*, pages 653–660. IOS Press, 2007.

[99] Bettina Krammer and Matthias S. Müller. MPI Application Development with MARMOT. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 893–900. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[100] Dieter Kranzlmüller, Christian Schaubschläger, Michael Scarpa, and Jens Volkert. A Modular Debugging Infrastructure for Parallel Programs. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, and Wolfgang V. Walter, editors, *Parallel Computing Software Technology, Algorithms, Architectures and Applications*, volume 13 of *Advances in Parallel Computing*, pages 143–150. North-Holland, 2004.

[101] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, and Todd Gamblin. Probabilistic Diagnosis of Performance Faults in Large-scale Parallel Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 213–222, New York, NY, USA, 2012. ACM.

[102] Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Anh, Martin Schulz, and Barry Rountree. Large Scale Debugging of Parallel Tasks

with AutomaDeD. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 50:1–50:10, New York, NY, USA, 2011. ACM.

[103] Leslie Lamport. Time Clocks, and the Ordering of Events in a Distributed System. *Commununications of the ACM*, 21:558–565, 1978.

[104] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit. Lessons learned at 208K: Towards Debugging Millions of Cores. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 26:1–26:9, Piscataway, NJ, USA, 2008. IEEE Press.

[105] Jae-Woo Lee, Leonardo R. Bachega, Samuel P. Midkiff, and Y. Charlie Hu. Ant: A Debugging Framework for MPI Parallel Programs. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, volume 7760 of *Lecture Notes in Computer Science*, pages 220–233. Springer Berlin Heidelberg, 2013.

[106] Soojung Lee. Fast, Centralized Detection and Resolution of Distributed Deadlocks in the Generalized Model. *IEEE Transactions on Software Engineering*, 30(9):561–573, 2004.

[107] Thomas Ludwig and Roland Wismüller. OMIS 2.0 – A Universal Interface for Monitoring Systems. In Marian Bubak, Jack Dongarra, and Jerzy Waśniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 267–276. Springer Berlin Heidelberg, 1997.

[108] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[109] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Mi-Young Park, Elizabeth Kleiman, Olga Weiss, Andre Wehe, and Melissa Yahya. The Importance of Run-Time Error Detection. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 145–155. Springer Berlin Heidelberg, 2010.

[110] Glenn R. Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Deadlock Detection in MPI Programs. *Concurrency and Computation: Practice and Experience*, 14:911–932, 2002.

[111] Olga Shumsky Matlin, Ewing Lusk, and William McCune. SPINning Parallel Systems Software. In Dragan Bošnački and Stefan Leue, editors, *Model Checking Software*, volume 2318 of *Lecture Notes in Computer Science*, pages 213–220. Springer Berlin Heidelberg, 2002.

[112] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In Michel Cosnard, Patrice Quinton, Michel Raynal, and Yves Robert, editors, *Parallel and Distributed Algorithms Conference*, pages 215–226. North-Holland, 1989.

[113] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 1.3. http://www.mpi-forum.org/docs/mpi-1.3/mpi-report-1.3-2008-05-30.pdf, 2008. Last visited on 09/12/2014.

[114] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012. Last visited on 09/12/2014.

[115] Shirley Moore, David Cronk, Kevin London, and Jack Dongarra. Review of Performance Analysis Tools for MPI Parallel Programs. In Yiannis Cotronis and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 241–248. Springer Berlin Heidelberg, 2001.

[116] Matthias S. Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng, and Carl Ponder. SPEC MPI2007 – An Application Benchmark Suite for Parallel Systems using MPI. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010.

[117] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller. A Framework for Scalable, Parallel Performance Monitoring. *Concurrency and Computation: Practice and Experience*, 22(6):720–735, 2010.

[118] Nicholas Nethercote and Julian Seward. Valgrind: A framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[119] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture—Programming Guide. http://docs.nvidia.com/cuda, 2014. Last visited on 09/12/2014.

[120] Patrick Ohly and Werner Krotz-Vogel. Automated MPI Correctness Checking: What if there was a magic option? In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*, 2007.

[121] Olga Papaemmanouil, Yanif Ahmad, Uğur Çetintemel, John Jannotti, and Yenel Yildirim. Extensible Optimization in Overlay Dissemination Trees. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 611–622, New York, NY, USA, 2006. ACM.

[122] Mi-Young Park, Su Jeong Shim, Yong-Kee Jun, and Hyuk-Ro Park. MPIRace-Check: Detection of Message Races in MPI Programs. In Christophe Cérin and Kuan-Ching Li, editors, *Advances in Grid and Pervasive Computing*, volume 4459 of *Lecture Notes in Computer Science*, pages 322–333. Springer Berlin Heidelberg, 2007.

[123] Yann Pouillon, Jean-Michel Beuken, Thierry Deutsch, Marc Torrent, and Xavier Gonze. Organizing Software Growth and Distributed Development: The Case of Abinit. *Computing in Science Engineering*, 13(1):62–69, 2011.

[124] Joachim Protze, Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. MPI Runtime Error Detection with MUST: Advanced Error Reports. In Alexey Cheptsov, Steffen Brinkmann, José Gracia, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2012*, pages 25–38. Springer Berlin Heidelberg, 2013.

[125] Joachim Protze, Tobias Hilbrich, Andreas Knüpfer, Bronis R. de Supinski, and Matthias S. Müller. Holistic Debugging of MPI Derived Datatypes. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 354–365. IEEE Computer Society, Washington, DC, USA, 2012.

[126] Joachim Protze, Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, Wolfgang E. Nagel, and Matthias S. Müller. MPI Runtime Error Detection with MUST: A Scalable and Crash-Safe Approach. In *To appear: 43nd International Conference on Parallel Processing (ICPP)*, Fifth International Workshop on Parallel Software Tools and Tool Infrastructures, Los Alamitos, CA, USA, 2014. IEEE Computer Society.

[127] Abtin Rahimian, Ilya Lashuk, Shravan Veerapaneni, Aparna Chandramowlishwaran, Dhairya Malhotra, Logan Moon, Rahul Sampath, Aashay Shringarpure, Jeffrey Vetter, Richard Vuduc, Denis Zorin, and George Biros. Petascale Direct Numerical Simulation of Blood Flow on 200K Cores and Heterogeneous Architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, 2010. IEEE Computer Society.

[128] Rogue Wave Software. Totalview Graphical Debugger. http://www.roguewave.com/products/totalview.aspx. Last visited on 09/12/2014.

[129] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, New York, NY, USA, 2003. ACM.

[130] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. Benchmarking the MRNet Distributed Tool Infrastructure: Lessons Learned. In *Proceedings of the 2004 IEEE 18th International Parallel and Distributed Processing Symposium*, IPDPS 2004, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[131] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Combining Static and Dynamic Validation of MPI Collective Communications. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 117–122, New York, NY, USA, 2013. ACM.

[132] Victor Samofalov, V. Krukov, B. Kuhn, S. Zheltov, Alexander V. Konovalov, and Jayant DeSouza. Automated Correctness Analysis of MPI Programs with Intel(r) Message Checker. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 901–908. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[133] Basile Schaeli and Roger D. Hersch. Dynamic Testing of Flow Graph Based Parallel Applications. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '08, pages 2:1–2:10, New York, NY, USA, 2008. ACM.

[134] Martin Schulz, Greg Bronevetsky, and Bronis R. de Supinski. On the Performance of Transparent MPI Piggyback Messages. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 194–201. Springer Berlin Heidelberg, 2008.

[135] Martin Schulz and Bronis R. de Supinski. A Flexible and Dynamic Infrastructure for MPI Tool Interoperability. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 193–202, Washington, DC, USA, 2006. IEEE Computer Society.

[136] Martin Schulz and Bronis R. de Supinski. PNMPI Tools: A Whole Lot Greater Than the Sum of Their Parts. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 30:1–30:10, New York, NY, USA, 2007. ACM.

[137] Subodh Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. A Sound Reduction of Persistent-Sets for Deadlock Detection in MPI Applications. In Rohit Gheyi and David Naumann, editors, *Formal Methods: Foundations and Applications*, volume 7498 of *Lecture Notes in Computer Science*, pages 194–209. Springer Berlin Heidelberg, 2012.

[138] Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. A Survey of MPI Related Debuggers and Tools. Technical Report UUCS-07-015, University of Utah School of Computing, 2007.

[139] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[140] Stephen F. Siegel. Using MPI-Spin to Model Check MPI Programs with Nonblocking Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI '06, 2006.

[141] Stephen F. Siegel. Model Checking Nonblocking MPI Programs. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 44–58. Springer Berlin Heidelberg, 2007.

[142] Stephen F. Siegel and George S. Avrunin. Modeling Wildcard-Free MPI Programs for Verification. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 95–106, New York, NY, USA, 2005. ACM.

[143] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2):10:1–10:34, 2008.

[144] Stephen F. Siegel and Timothy K. Zirkel. Automatic Formal Verification of MPI-Based Parallel Programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 309–310, New York, NY, USA, 2011. ACM.

[145] William C. Skamarock, Joseph B. Klemp, Jimy Dudhia, David O. Gill, Dale M. Barker, Wei Wang, and Jordan G. Powers. A description of the Advanced Research WRF Version 2. Technical report, National Center for Atmospheric Research, 2008.

[146] Standard Performance Evaluation Corporation. SPEC MPIL2007 Result: Endeavor. http://www.spec.org/mpi/results/res2012q1/mpi2007-20120306-00389.html. Last visited on 09/12/2014.

[147] Standard Performance Evaluation Corporation. SPEC MPIL2007 Result: SGI Altix ICE 8200EX. http://www.spec.org/mpi/results/res2010q1/mpi2007-20100119-00199.html. Last visited on 09/12/2014.

[148] Anne M. Stark. Bug repellent for supercomputers proves effective. https://www.llnl.gov/news/newsreleases/2012/Nov/NR12-11-02.html, 2012. Last visited on 09/12/2014.

[149] Vaidy S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[150] Rajeev Thakur and William Gropp. Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 46–55. Springer Berlin Heidelberg, 2007.

[151] The Multicore Association. Multicore Communications API Working Group (MCAPI). http://www.multicore-association.org/workgroup/mcapi.php. Last visited on 09/12/2014.

[152] Top500. Top 500 Supercomputer Sites. http://www.top500.org. Last visited on 09/12/2014.

[153] Jesper Larsson Träff and Joachim Worringen. Verifying Collective MPI Calls. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 18–27. Springer Berlin Heidelberg, 2004.

[154] Jesper Larsson Träff and Joachim Worringen. The MPI/SX Collectives Verification Library. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 909–916. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[155] Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 66–79. Springer Berlin Heidelberg, 2008.

[156] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 285–286, New York, NY, USA, 2008. ACM.

[157] Michael L. Van de Vanter, Douglass E. Post, and Mary E. Zosel. HPC Needs a Tool Strategy. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 55–59, New York, NY, USA, 2005. ACM.

[158] Jeffrey S. Vetter and Chris Chambreau. mpiP: Lightweight, Scalable MPI Profiling. `http://mpip.sourceforge.net/`, 2014. Last visited on 09/12/2014.

[159] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[160] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[161] Anh Vo and Ganesh Gopalakrishnan. Scalable Verification of MPI Programs. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, IPDPSW '10, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[162] Anh Vo, Ganesh Gopalakrishnan, Robert M. Kirby, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. Large Scale Verification of MPI Programs Using Lamport Clocks with Lazy Update. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 330–339, Washington, DC, USA, 2011. IEEE Computer Society.

[163] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[164] William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[165] Michael Wagner, Tobias Hilbrich, and Holger Brunst. Online Performance Analysis: An Event-based Workflow Design Towards Exascale. In *To appear: The 16th IEEE International Conference on High Performance Computing and Communications*, HPCC 2014, 2014.

[166] Michael Wagner, Andreas Knüpfer, and Wolfgang E. Nagel. Enhanced Encoding Techniques for the Open Trace Format 2. *Procedia Computer Science*, 9:1979–1987, 2012.

[167] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. AM++: A Generalized Active Message Framework. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 401–410, New York, NY, USA, 2010. ACM.

[168] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active Pebbles: Parallel Programming for Data-Driven Applications. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 235–245, New York, NY, USA, 2011. ACM.

[169] Roland Wismüller, Thomas Ludwig, Wolfgang Karl, and Arndt Bode. Monitoring Concepts for Parallel Systems - An Evolution Towards Interoperable Tool Environments. *Scalable Computing: Practice and Experience*, 4(3), 2001.

[170] Roland Wismüller, Jörg Trinitis, and Thomas Ludwig. OCM–A Monitoring System for Interoperable Tools. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*, pages 1–9. ACM Press, 1998.

[171] Xing Wu and Frank Mueller. ScalaExtrap: Trace-Based Communication Extrapolation for SPMD Programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 113–122, New York, NY, USA, 2011. ACM.

[172] Hyun Joong Yoon and Doo Yong Lee. Deadlock-Free Scheduling of Photolithography Equipment in Semiconductor Fabrication. *IEEE Transactions on Semiconductor Manufacturing*, 17(1):42–54, 2004.

[173] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[174] Yuan Zhang and Evelyn Duesterwald. Barrier Matching for Programs with Textually Unaligned Barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 194–204, New York, NY, USA, 2007. ACM.

Glossary

analysis	Also *[tool] analysis*. A part of the functionality of a runtime tool. In the proposed tool infrastructure abstraction, an analysis is a piece of tool functionality that can be triggered by an event and that is implemented in a module.
back-end	In the MRNet abstraction, code that executes on the leaves of a TBON layout.
centralized	A [tool] architecture type that uses a single process or thread for some computation.
channel identifier	For events that a place handles, a *channel identifier* can identify a descendant place that created this event.
check	A correctness analysis of a correctness tool.
defect	Commonly *bug*. Defects in the source code can result in an *infested*—incorrect—application state and become visible as failures.
event	Occurrence of something observable. In the proposed tool infrastructure abstraction, the data that results when a place triggers an instrumented hook.
event aggregation	A computation that occurs on places, which uses events as inputs, and that replaces the input events with new event(s).
event-flow	In the proposed tool infrastructure abstraction, if a place observes or receives an event, the *event-flow* provides a set of definitions that specify how the event must be handled.
fan-in	The number of child places of a place in a topology (assuming a parent-child relation as in a tree topology).
front-end	In the MRNet abstraction, code that executes on the root of a TBON layout.
GTI	*Generic Tools Infrastructure* The prototype implementation that implements the proposed tool infrastructure abstraction.
hook	In the proposed tool infrastructure abstraction, a source of events, which can be instrumented.
HPC	*High Performance Computing* Computing on a compute system that drastically outperforms current desktop systems. This thesis primarilly addresses distributed memory systems, where compute cores cannot access all available memory. The use of of a parallel programming paradigm such as provided by MPI allows an application to utilize such a system.
interleaving	Without synchronization, events on parallel processes or threads can occur in different orders for different executions of an application. The term *interleaving* refers to one such execution with a specific event timing.
layer	In the proposed infrastructure abstraction, a set of places.
match layer	A specifically marked layer for the proposed runtime correctness concepts to which the *P2PMatch* and *TransitionSystem* modules are mapped.

module	A reusable piece of a runtime tool. In the proposed tool infrastructure abstraction, a collection of *analyses* and their associated data.
MPI	*Message-Passing Interface* The MPI standard defines a set of functions that enable message-based data transfers between processes with distributed memory. The standard uses a single executable that is executed by all processes, however, each process differs by a unique identifier called *rank*.
MUST	*Marmot Umpire Scalable Tool* The prototype implementation that implements the proposed distributed MPI runtime MPI verification concepts as a GTI-based tool.
operation	An invocation of an MPI function. In Chapter 4 as part of the proposed infrastructure abstraction, a computation that can transform arguments of a hook.
place	A thread or process that is part of an application with an attached runtime tool. *Tool places* serve as an additional process or thread to offload computations of a tool, *application/leaf places* are the processes or threads to which the tool connects, and a *root place* is a tool place that has all application places as its descendants.
precise	In the context of runtime correctness tools, a tool is precise if its detection capabilities do not use simplifications or heuristics that can cause false positives or false negatives.
protocol	Also *communication protocol*. A tool component of a GTI-based tool that provides a means to transfer events from a place to a connected place.
rank	An identifier for a process of an MPI parallel application.
resource	In the context of MPI, information on MPI handles such as communicators, process groups, and requests.
runtime verification	Often also *correctness checking*, *runtime checking*, *dynamic verification*, or *dynamic testing*. An approach to detect software defects with information that was captured during the execution of an application.
slowdown	The runtime of an application run with a runtime tool divided by the runtime of a reference run. Often calculated from a specific timespan of an execution to exclude initialization/finalization overheads.
specification	In the proposed infrastructure abstraction, *specifications* describe a runtime tool.
strategy	Also *communication strategy*. A tool component of a GTI-based tool that times the use of a communication protocol.
tag	A message identifier for MPI-based point-to-point communication.
TBON	*Tree-Based Overlay Network* A rooted tree network that spans all application processes/threads as its leaves.
tool infrastructure	An infrastructure or a framework that provides services for a runtime tool.
trace	Lists of unprocessed events for the proposed distributed transition system for MPI deadlock detection.
wave	A set of associated events that originate from different places.
weaver	Part of the GTI prototype that processes a tool specification to instantiate a GTI-based tool.

WFG	*Wait-For Graph* A graph that represents wait-for conditions of an execution state of a parallel application. The graph-based deadlock search in this thesis uses the AND⊕OR WFG that provides two types of wait-for dependencies.
wrapper	A part of a runtime tool that observes events.

List of Figures

List of Tables

List of Symbols

Common Terms

p	Process count for a parallel application
$P = 0, 1 \ldots, p-1$	Set of process identifiers (ranks)
KiB	Binary version of kilobytes, i.e., 1024 bytes
GHz	Gigahertz

Analysis Specification

A	Set of analyses
$M = M_A \cup M_T$	Set of tool modules
M_A	Set of aggregation modules
M_T	Set of analysis modules
$\text{arity} : A \cup O \cup H \to \mathbb{N}$	Function that specifies the arrity for analyses, operations, and hooks
$a_M : M \to \mathcal{P}(A)$	Function that assigns analyses to modules
$d_M : M \to \mathcal{P}(M_T)$	Function that specifies module dependencies
$s_A : M_T \to \mathcal{P}(M_A)$	Function that specifies which aggregation modules an analysis module supports
O	Set of operations

Hook Specification

H	Set of hooks
$\text{dir} : H \to \{\text{primary}, \text{broadcast}, \text{intralayer}\}$	Function that assigns hooks a communication direction
$\text{AH} \subseteq A \times H$	Set of analysis-hook mappings
$m_{A,H} : \text{AH} \to \bigcup_{i,j \in \mathbb{N}} \text{ASeqs}_{i,j}$	Function that associates each analysis-hook mapping with an argument mapping (which may include operation mappings in turn)

Layout Specification

L	Set of layers
$\mathcal{L} = (L, E_\mathcal{L} \subseteq L \times L)$	Layer tree where $\mathcal{L}$ is an arboresence with root $l_0 \in L$
$\text{size} : L \to \mathbb{N} \setminus \{0\}$	Function that associates layers with a place count
$\text{bsize} : L \to \mathbb{N} \setminus \{0\}$	Function that associates layers with a blocksize for block-distribution
$m_\text{com} : E_\mathcal{L} \to M_P \times M_S$	Function that assigns pairs of protocol and strategy modules to each connection of the layer tree
$m_\text{place} : L \setminus \{l_0\} \to M_D$	Function that assigns a place module to each layer
$L_\text{intra} \subseteq L \setminus \{l_0\}$	Set that specifies which layers use intralayer communication
$m_\text{icom} : L_\text{intra} \to M_P \times M_S$	Function that assigns communication module pairs to layers that use intralayer communication

$m_{L,M} : L \to \mathcal{P}(M_T)$	Function that assigns modules to layers (module-layer mapping)

GTI Specification

M_S	Set of strategy modules
M_P	Set of protocol modules
M_D	Set of place modules that provide drivers to tool places

A GTI: A Parallel Tools Infrastructure

A.1 Module Relationships

GTI organizes instances of modules in a DAG to simplify instance creation and destruction. Each module receives access to interfaces of their dependent module instances. For analysis and aggregation modules (modules in M), the function d_M specifies this relationship. Arrival, wrapper, strategy, protocol, and place modules use predefined module dependencies. These predefined dependencies depend on the tool layout and the module placement and are:

Place instances depends on:

- 1 Strategy module (ancestor layer),
- 0–N Strategy modules (descendant layers),
- 0–1 Strategy module (intralayer, see Section 4.3.4 on page 58),
- 1 Arrival module, and
- 1 Wrapper module.

Wrapper instances depends on:

- 0–N Strategy modules (ancestor layer),
- 1 Strategy modules (descendant layers),
- 0–1 Strategy modules (intralayer, see Section 4.3.4 on page 58), and
- M Analysis/Aggregation modules.

Arrival instances depends on:

- 0–N Strategy modules (towards root direction),
- 1 Strategy modules (from-leaves direction), and
- M Analysis/Aggregation modules.

Strategy instances depends on:

- 1 Protocol module.

Protocol instances have no dependencies.

GTI uses a P^{n}MPI configuration file to provide arguments to module instances. These arguments specify whether and how often a module uses an optional dependency. As an example, a place module receives information on the number of strategy modules (0–N) that connect the place to descendant layers. Figure A.1 presents dependencies between module instances for the GTI mappings and layout from Figure 4.3 (page 40). The large boxes distinguish module instance hierarchies for layers l_0–l_2. Boxes with rounded edges represent module instances and their labels indicate the module type. Arrows indicate module dependencies where the arrow points towards the dependent module. Finally, pointed lines between protocol modules indicate communication connections across layers. Note that except for the two tool modules m_0 and m_1 the figure only indicates module types rather than module names. Thus, the two strategy module instances of layer l_1 may be instances of the same or of distinct modules. The root module on application places is a single wrapper instance, which GTI creates when it perceives the first event. On tool places, a single place module instance serves as root instance, which GTI creates when it starts the place.

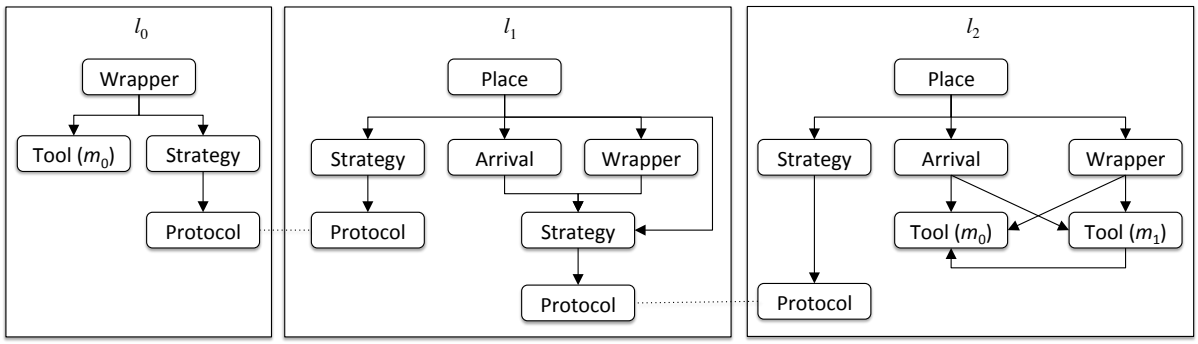


Figure A.1: *Example module instance hierarchy for the example from Figure 4.3.*

A.2 Channel Tree Algorithm Example

As an example to illustrate the outcome of the channel tree algorithms and the channel trees that they create, consider node $T_{2,0}$ in Figure 4.15(a) (page 70). The listing below summarizes key actions of the tool driver (Figure 4.10 on page 62) and its queries and updates to the channel tree along with their return values:

(1)
- Actions:
  - $\text{findEventToDequeue}(\epsilon) = (\bot, \text{NULL}, \epsilon)$
  - Receive b_2 with channel identifier 1.0
  - $\text{treeNodeAncestorOrDescendantSuspended}(1.0) = \bot$
  - Process b_2
- Resulting channel tree: Figure A.2(a)

(2)
- Actions:
  - $\text{findEventToDequeue}(\epsilon) = (\bot, \text{NULL}, \epsilon)$
  - Receive $a_{0,1}$ with channel identifier 0
  - $\text{treeNodeAncestorOrDescendantSuspended}(0) = \bot$
  - Process $a_{0,1}$
  - $\text{suspendChannel}(0)$
- Resulting channel tree: Figure A.2(b)

(3)
- Actions:
  - $\text{findEventToDequeue}(\epsilon) = (\bot, \text{NULL}, \epsilon)$
  - Receive c_1 with channel identifier 0.1
  - $\text{treeNodeAncestorOrDescendantSuspended}(0.1) = \top$
  - $\text{enqueue}(c_1, 0.1)$
- Resulting channel tree: Figure A.2(c)

(4)
- Actions:
  - $\text{findEventToDequeue}(\epsilon) = (\bot, \text{NULL}, \epsilon)$
  - Receive c_0 with channel identifier 0.0
  - $\text{treeNodeAncestorOrDescendantSuspended}(0.0) = \top$
  - $\text{enqueue}(c_0, 0.0)$

- Resulting channel tree: Figure A.2(d)

(5)
- Actions:
  - $\text{findEventToDequeue}(\epsilon) = (\bot, \text{NULL}, \epsilon)$
  - Receive $a_{2,3}$ with channel identifier 1
  - $\text{treeNodeAncestorOrDescendantSuspended}(1) = \bot$
  - Process $a_{2,3}$
  - $\text{openChannel}(0)$
- Resulting channel tree: Figure A.2(e)

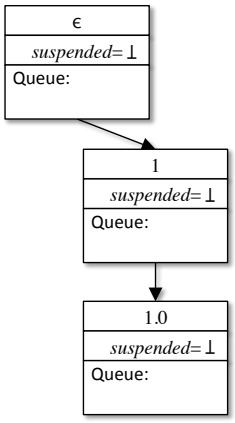
(6)
- Actions:
  - $\text{findEventToDequeue}(\epsilon) = (\top, c_0, 0.0)$
  - Process c_0

(7)
- Actions:
  - $\text{findEventToDequeue}(\epsilon) = (\top, c_1, 0.1)$
  - Process c_1
- Resulting channel tree: Figure A.2(f)

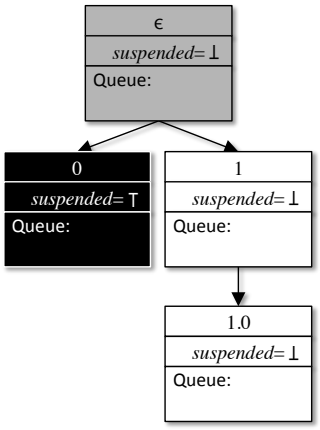
(8)
- Actions:
  - $\text{findEventToDequeue}(\epsilon) = (\bot, \text{NULL}, \epsilon)$
  - Receive c_3 with channel identifier 1.1
  - $\text{treeNodeAncestorOrDescendantSuspended}(1.1) = \bot$
  - Process c_3
- Resulting channel tree: Figure A.2(g)

The listing assigns iterations of the algorithm in Figure 4.10 numbers and assumes that each of these iterations uses the *regular* direction, while the input communication strategy always provides an event to receive, i.e., no idle loops. While the listing specifies the individual channel tree queries and updates, Figure A.2 presents the state of the channel tree after most of the iterations. The figure highlights suspended nodes in black and white, while it also highlights nodes whose process index sets overlap with suspended or non-empty queue nodes in gray.

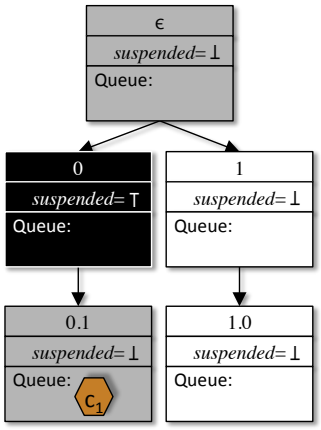
Iteration (1) receives and processes event b_2. Afterwards, iteration (2) receives and processes event $a_{0,1}$. Since the processing starts an aggregation (that waits for $a_{2,3}$), the driver algorithm suspends the channel tree node for this event. Iterations (3) and (4) then receive events c_1 and c_0, that the driver algorithm enqueues since the process index sets that their channel identifiers represent overlap with the set of the suspended node. When the driver receives and processes event $a_{2,3}$ in iteration (5), it finishes the aggregation and removes the suspension for channel identifier 0. Iterations (6) and (7) then dequeue and process events c_0 and c_1. Finally, iteration 8 receives and processes event c_3.



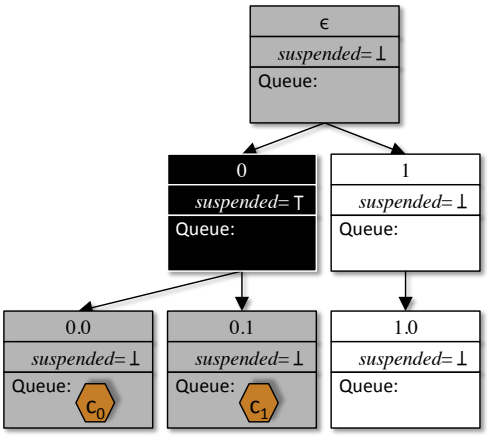
(a) Added node for channel identifier 1.0.



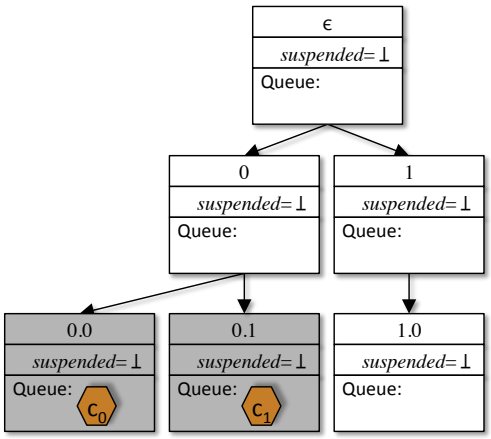
(b) Added and suspended node for channel identifier 0.



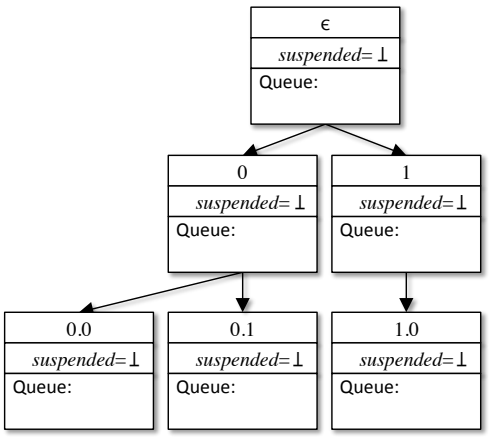
(c) Added node for channel identifier 0.1 and added an event to its queue.



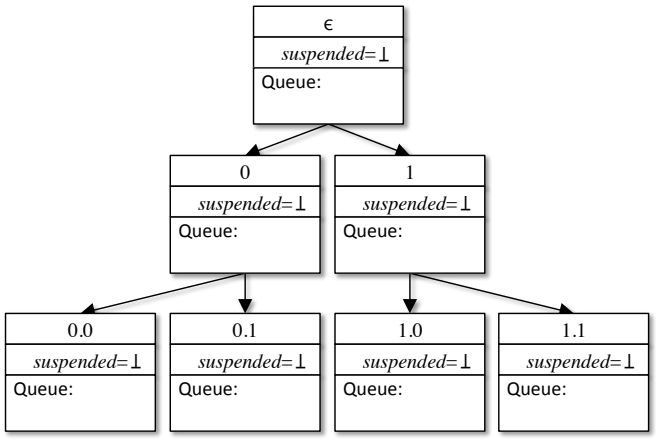
(d) Added node for channel identifier 0.0 and added an event to its queue.



(e) Removed suspension to node associated with channel identifier 0.



(f) Dequeued all events.



(g) Added node for channel identifier 1.1.

Figure A.2: *Series of channel tree states for an execution of node $T_{2,0}$ from Figure 4.15.*

B Detailed MUST Designs

B.1 Parallel and Location Identifiers in MUST

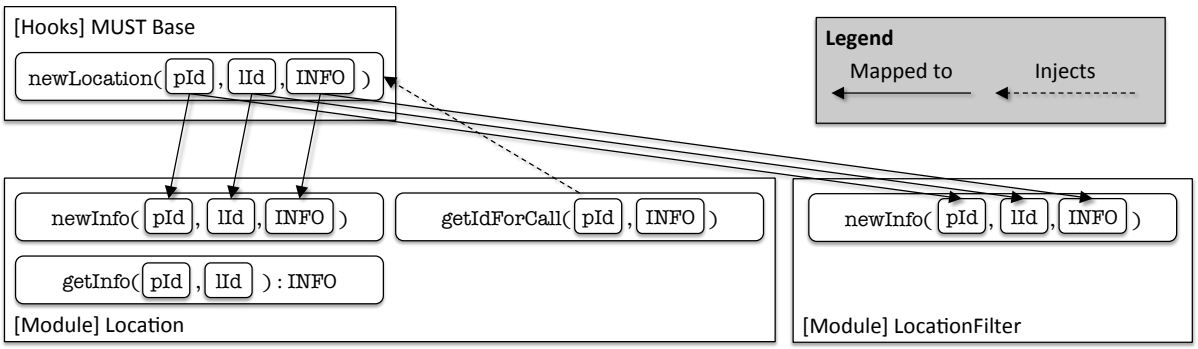


Figure B.1: *Analyses of the* Location *module and the hooks that serve for the forwarding of location information.*

Some modules require information on MPI ranks to discern the events that trigger their analyses. MUST provides parallel identifiers (pIds) to associate rank information with events. The pId includes the MPI rank, but could also include additional identifiers, e.g., a thread number to distinguish threads on a process (if an application uses MPI in combination with a threading paradigm like OpenMP). A GTI operation creates the pId of an event directly on the application processes. MUST modules use a dependency to a module from the identifier package to retrieve information from pIds, e.g., to retrieve an MPI rank from a pId.

A location identifier (lId) provides information on call locations to MUST modules. Information on call locations includes the name of the function that created an event and, if available, a call stack[1]. Since an integer identfier cannot store an arbitrary string, nor a call stack, MUST maps $(\mathrm{pId}, \mathrm{lId})$ pairs to information structures that specify the call name and call stack (if available). MUST provides the *Location* module to create lIds and to manage the identifier mappings. Figure B.1 summarizes the analyses and services of the *Location* module that manages and provides lIds. The figure uses the module-hook representation that Section 5.2 on page 85 introduces. The `getIdForCall` analysis of the *Location* module provides location identifiers for a description of the MPI call (and stack) as well as a pId, The *Location* module compares the given pId and call site information (INFO) to its mapping structures to either return the lId of an existing similar mapping, or it creates a new lId that extends the existing mappings. Whenever this analysis introduces a new lId, i.e., recognizes a new call site, then MUST distributes the new $(\mathrm{pId}, \mathrm{lId})$ pair and its call site information to all descendant layers. The *Location* module distributes this information with an injected event to which the module is mapped itself. The `newLocation` hook serves for this event injection.

The *Location* module supports the aggregation module *LocationFilter* that removes information on redundant call sites. This filter module detects whether distinct MPI ranks use similar lIds for similar call sites. If so, a `newLocation` event describes redundant information and qualifies for filtering.

Finally, MUST modules use a dependency to the *Location* module to translate $(\mathrm{pId}, \mathrm{lIds})$ pairs into call site information. The `getInfo` service (analysis without a mapping) provides this information. This dependency ensures that GTI places the *Location* module onto all layers that require this module. In turn,

[1] The MUST implementation can use an installation of the Stackwalker API of the Dyninst project (http://www.dyninst.org/) to retrieve this information.

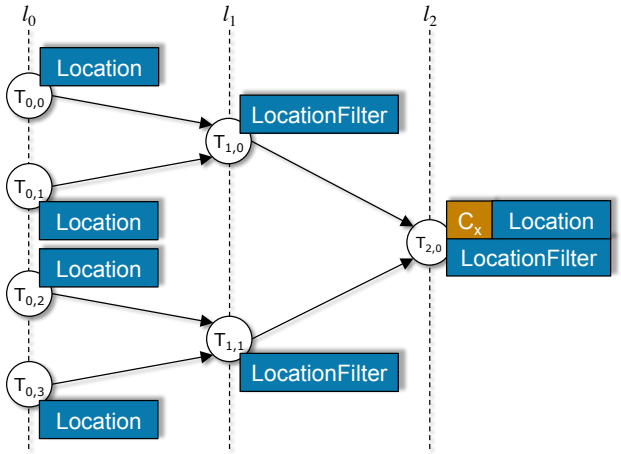


Figure B.2: *Example layout that illustrates the placement of the* Location *module and its filter module (*LocationFilter).

the presence of the module on a layer ensures that a tool instance forwards events of the `newLocation` hook towards the layer, due to the mapping of the `newInfo` analysis to the hook.

Figure B.2 illustrates the placement of the *Location* and *LocationFilter* modules with four application processes and a single check module (C_x) that is placed onto layer l_2. The application layer (l_0) uses the *Location* module to create lIds and layer l_2 uses this module to provide information on call sites to module C_x. The presence of the *Location* module on layer l_2 allows GTI to automatically place the aggregation module *LocationFilter* onto layers l_1 and l_2.

B.2 Correctness Message Logging in MUST

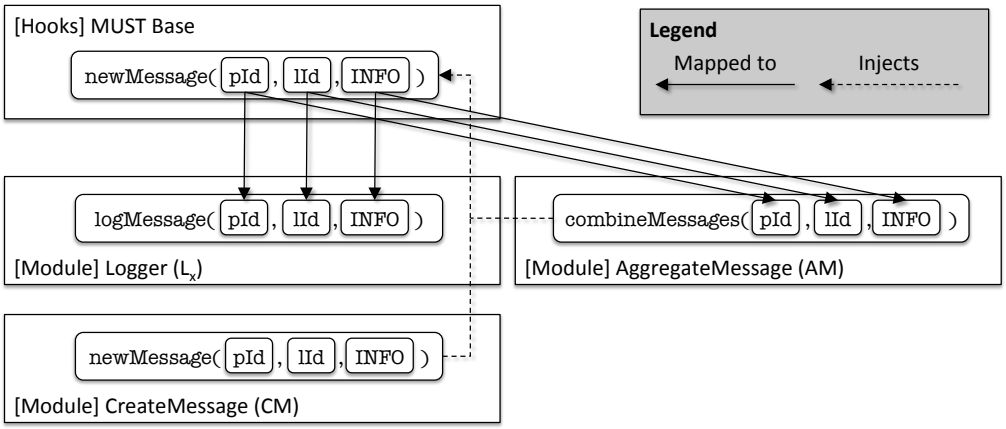


Figure B.3: *Modules of MUST's logging system and the hooks that is used to forward information.*

An efficient logging system should not provide one correctness report per application process, but rather a single report with condensed information. As a result, MUST provides a flexible logging system that can combine similar correctness messages. Figure B.3 presents the modules of MUST's logging package and the hooks that inject events for logging purposes.

Modules use a dependency to the *CreateMessage* module (CM) to inject events for correctness messages. The `newMessage` analysis (without a mapping) provides an interface to describe a new message. When a module issues this analysis, the *CreateMessage* module does not store the new message in a log directly, but rather injects an event that describes the message. The module uses the `newMessage` hook

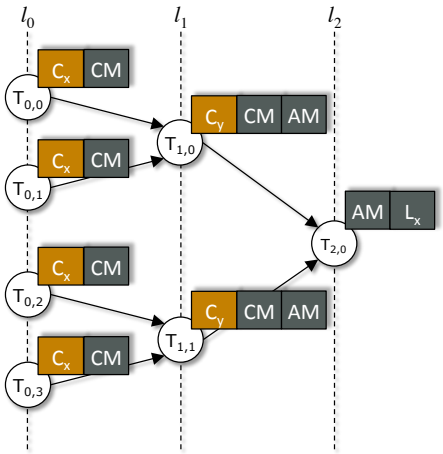


Figure B.4: *Example layout that illustrates a logging module placement.*

for this purpose. This design enables the use of distinct and use-case-specific modules to log messages. MUST uses the term *Logger* for any module that handles, stores, or visualizes messages. These modules provide an analysis that is mapped to the `newMessage` hook, such as the example module in Figure B.3. Thus, a *Logger* module on a layer that consists of a single place, can provide a single correctness report for all application processes. At the same time, different *Logger* placements allow use-case-specific logging designs.

The *AggregateMessage* module (AM) is an aggregation module that aids *Logger* modules. The aggregation condenses similar correctness messages to reduce the number of events that the tool communicates and to provide short reports. As an example, if all application processes exhibit a similar MPI usage error, then the *AggregateMessage* module ensures that *Loggers* do no log individual correctness messages for all processes, but rather a single report that states that all processes exhibit the failure.

Figure B.4 illustrates the usage of the logging package modules with an example instantiation. The scenario uses four application processes, a check module (C_x) on layer l_0, and a check module (C_y) on layer l_1. In addition, it uses a the *Logger* module L_x on layer l_2. GTI places the *CreateMessage* onto layers l_0 and l_1, since the check modules C_x and C_y depend upon this module. L_x supports the *AggregateMessage* aggregation, thus, GTI places this module onto layers l_1 and l_2.

B.3 Status Source Updates for the *P2PMatch* Module

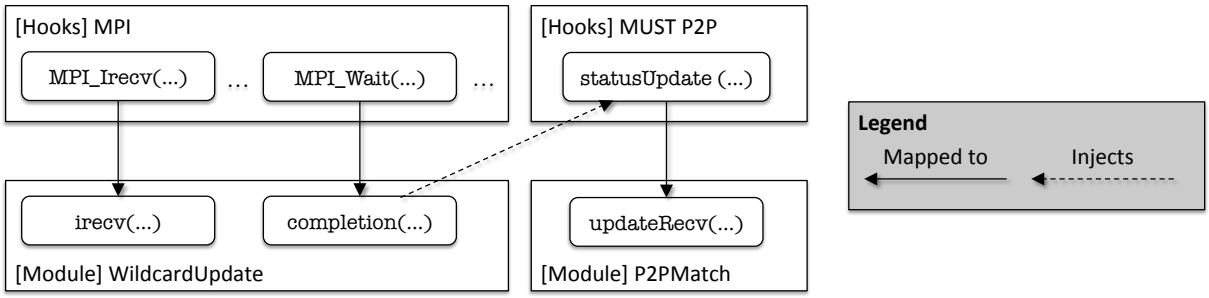


Figure B.5: *The* WildcardUpdate *module as a filter for receive operation updates of the* P2PMatch *module.*

Section 5.2 on page 85 describes that the *P2PMatch* module for point-to-point message matching requires information on completed wildcard receives. The module uses matching information from the MPI implementation to adjust its own matching decisions accordingly. MPI provides information on

its matching decissions when a blocking receive operation completes or when a completion operation successfully handled an MPI request. Direct analysis mappings to the hooks that procvide this information, e.g., to `MPI_Wait`, would create superfluous events since these operations provide this data for all point-to-point operations. The *P2PMatch* module, however, only requires this data for wildcard receives.

MUST uses the *WildcardUpdate* module to implement a conditional analysis mapping. Its default layout places the module on the application layer to only communicate necessary events towards further layers. Figure B.5 sketches the design of this module and the hook that forwards matching information for wildcard receives. The *WildcardUpdate* module requires information on which MPI requests are associated with nonblocking receive operations in order to apply its conditional filtering. Analysis mappings to all operations that initiate nonblocking receives, e.g., `MPI_Irecv` or `MPI_Start`, provide this information. The module then stores all requests of wildcard receive operations in a map data structure. Analysis mappings of the *WildcardUpdate* module provide all MPI matching information for completion operations such as `MPI_Wait` and for all blocking receive operations such as `MPI_Recv`. For blocking receives the module injects an event with the `statusUpdate` hook if the receive uses a wildcard source, as well as for completions that successfully complete a requests that is in the module's map data structure. The implementation of MUST uses multiple hooks and analyses to handle distinct types of completion operations and to provide updates for multiple wildcard receives with a single event.

The *P2PMatch* module provides the `updateRecv` analysis, which MUST maps to the `statusUpdate` hook. Thus, GTI only forwards matching information for completed wildcard receives to the module.

B.4 Collective Operation Premise Exchange for the *TransitionSystem* Module

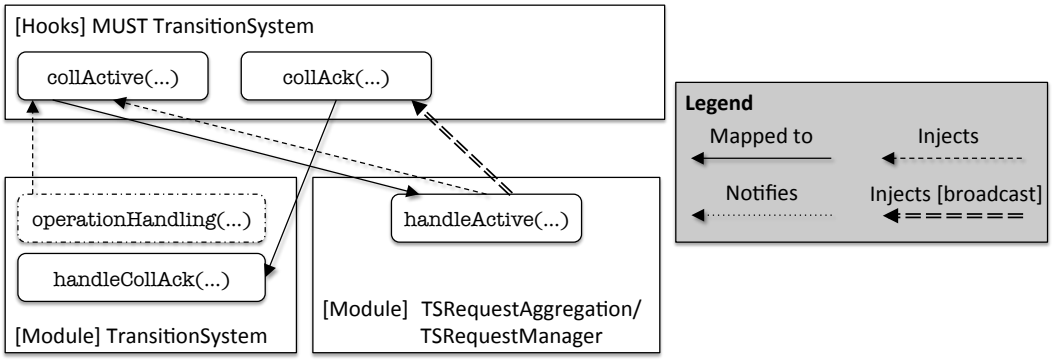


Figure B.6: *Modules to scalably handle premise exchanges for the collective handling of the* TransitionSystem *module.*

The *TransitionSystem* module from Section 5.4.5 (page 102) uses the `collActive` hook to exchange premise information on transition system rules that apply to collective operations. Depending on the process group of a collective, these exchanges may involve all or most places of the match layer. The `collActive` hook uses the primary communication direction. An aggregation module *TSRequestAggregation* combines events of this hook if they belong to the same collective. Figure B.6 illustrates this module and its analysis-hook mappings as a module-hook chart. MUST layouts place the *TSRequestManager* module onto the layer of the root place to both enable the aggregation module and to evaluate when all processes joined a collective, i.e., when all *TransitionSystem* module instances (that receive at least one operation of the collective) injected a `collActive` event for a certain collecitve. If so, the *TSRequestManager* module injects an acknowledgement event with the `collAck` hook (broadcast direction) to notify all *TransitionSystem* module instances that they may apply the transition rule to the operations of the specified collective.

B.5 Synchronization Between *TransitionSystem* Module Instances

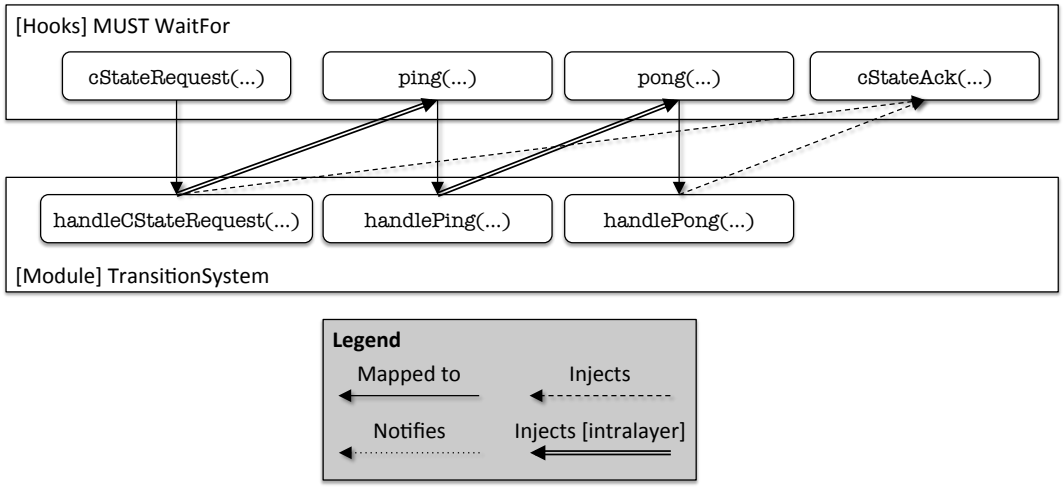


Figure B.7: *Hooks and analyses to synchronize* TransitionSystem *module instances during a request for a consistent state.*

The synchronization of the *TransitionSystem* module from Section 5.5 (page 108) uses a ping-pong communication between pairs of match layer places. Figure B.7 illustrates the hooks that inject the respective events and the analyses that apply to these hooks. The `handleCStateRequest` of a *TransitionSystem* module instance injects `ping` events for each place that could host a receive operation for an active send operation on the instance. These events use the intralayer direction and the `handlePing` analysis receives them. The latter analysis immediatly injects a further intralayer event with the `pong` hook. Finally, the *handlePong* analysis receives the replies to the `ping` events and adapts a count of outstanding replies. If all replies arrived, or no ping-pong communication is necessary, then a *TransitionSystem* module instance injects an event with the `cStateAck` hook.

C Supplemental Measurement Results

C.1 Collective Kernel Overheads of the *TransitionSystem* Module

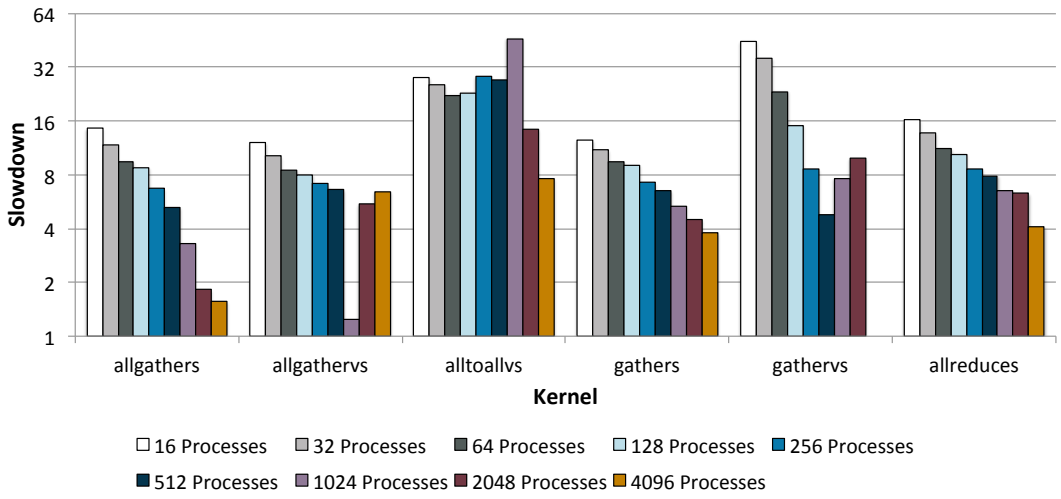


Figure C.1: *Overhead of the* TransitionSystem *layout on Sierra with fan-in* 4 *for representatives of the collective classes from Table 5.1.*

Figure C.1 presents slowdowns for the *TransitionSystem* layout with kernels that represent the collective classes from Table 5.1 (page 89). The measurement compares to the evaluation of the *CollectiveMatch* layout in Section 6.2.2 (page 121). The *TransitionSystem* module increases the workload for the analysis of the collective operations. However, while the *CollectiveMatch* module may require a different handling for the each collective, the *TransitionSystem* module uses the same handling for all collectives. In addition, the change in the communication strategies increases the communication cost for the *TransitionSystem* layout. All measurements use a fan-in of 4.

For 4,096 processes, the *CollectiveMatch* layout exhibits slowdowns of 1 to 2 for the kernels *allgathers*, *gathers*, *gathervs*, and *allreduces*. The *TransitionSystem* layout causes higher overheads in a range of about 2 to 4. Slowdowns for the *gathervs* kernel increase for 1,024 and 2,048 processes. The immediate communication strategy of the layout yields high numbers of small intralayer messages that communicate type matching information for this benchmark. In addition, runtimes increase unexpectedly high for 4,096 processes even in the reference run (thus no result for this scale). These results suggest that future improvements of the immediate intralayer communication strategy could yield performance improvements, especially since GTI's profiling data reports high idle times for this kernel.

The kernels *allgathervs* and *alltoallvs* behave similar as for the *P2PMatch* layout where the dip at 1,024 processes for the former kernel is likely an outlier. As for the *CollectiveMatch* layout, the decreasing iteration counts of the *alltoallvs* kernel yield lower slowdowns at 2,048 and 4,096 processes.

Overall, the slowdowns in Figure C.1 are noticable but do not highlight increases with scale (except for the *gathervs* kernel). In addition, overheads are far lower than point-to-point handling overheads (see Section 6.2.1.1 on page 117).

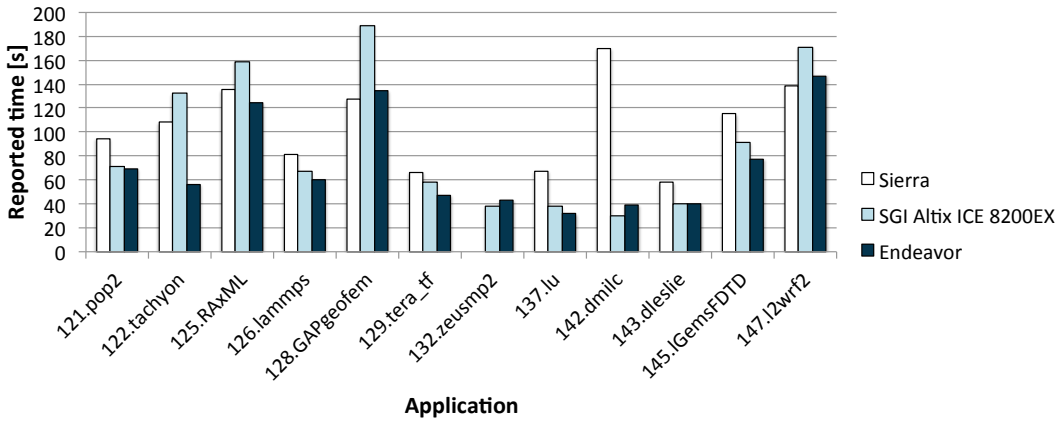


Figure C.2: *Comparison of reported SPEC MPI2007 benchmark times for Sierra and two further Systems with Intel Xeon CPUs for* 2,096 *processes.*

C.2 SPEC MPI2007 Reference Time Comparison

Figure C.2 compares SPEC MPI2007 result for 2,048 processes with three different compute systems. The results include all applications that support the *lref* data set. The goal of this comparison is to relate result times on Sierra with published results for comparative systems. The comparison uses two systems with Intel CPUs that have an about identical CPU speed (2.6GHz to 2.8GHz) since no reported SPEC results exist for a similar architecture as Sierra (same CPUs and same interconnect). All three systems use an QDR InfiniBand interconnect. The first reported result is from an SGI system [147] and the second from the Endeavor system [146].

While hardware differences as well as software differences impact the comparability of the three systems, a mayor difference is the compute node allocation. The results for both the SGI system and the Endeavor system place MPI ranks onto a toplogically compact set of compute nodes that uses a minimal amount of switches. The results on the Sierra system use batch allocations during regular system operation. Thus, the allocated nodes on Sierra can be distributed within the comparatively large cluster. Studies [18] for such operation modes underline that job placement as well as the presence of other jobs can have a large impact on application performance.

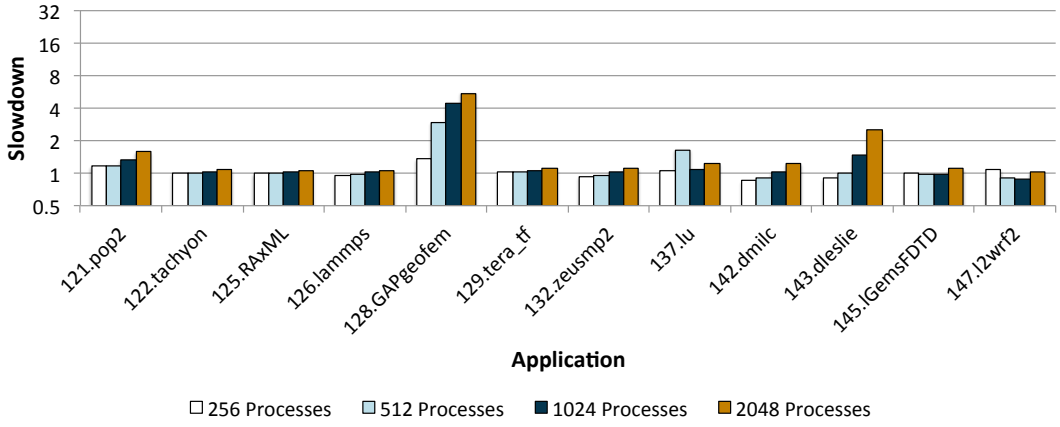
Thus, the comparison does not serve to benchmark the Sierra system, but rather to detect whether benchmarks show unexpectedly long runtimes. Using the maxiumum reported time from the two comparison systems, Sierra exhibitis 33% longer runtime for *121.pop2*, 76% longer runtime for *137.lu*, 332% longer runtime for *142.dmilc*, 44% longer runtime for *143.dleslie*, and 25.7% longer runtime for *145.lGemsFDTD*. At the same time, the Sierra system provides runtimes that are within the comparsion times—or even lower—for four of the benchmarks. Runtime increase of 44% provide a potential for decreased tool slowdowns in Section 6.3 (page 124). However, the the effects that cause such runtime increases can impact the tool itself and thus do not necessarily provide an advantage for the measurements. On the other hands the high runtime increases for the applications *137.lu* and *142.dmilc* suggest that tool slowdowns for these applications could differ substantially on other systems. Finally, *132.zeusmp2* fails to validate starting at 1,024 processes on Sierra. While the application appears to provide correct results, a single line of the application output diverges from the expected output. This behavior remains a target for future study. The slowdown calculations in Section 6.3 use the overal runtime of the *runspec* tool for this benchmark instead.

C.3 Slowdowns with Application Crash Handling for SPEC MPI

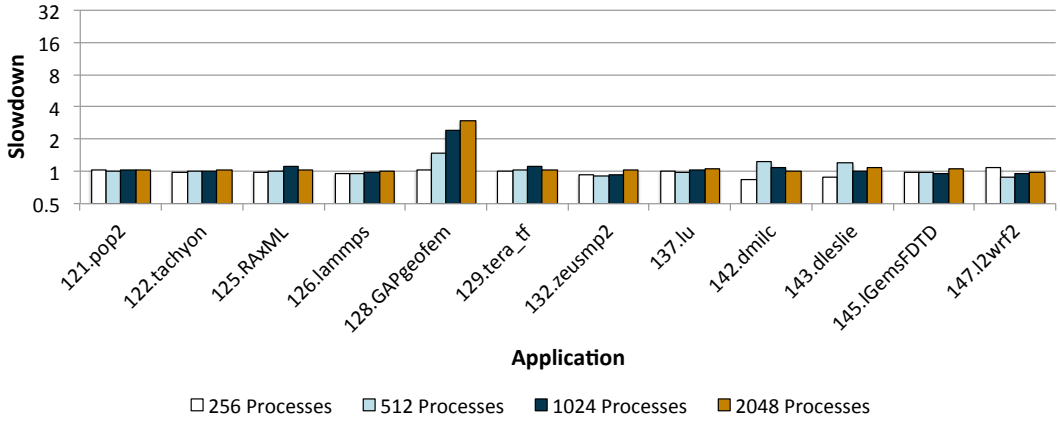
Figure C.3 presents slowdowns of the *P2PMatch*, *CollectiveMatch*, and *TransitionSystem* layouts for SPEC MPI2007. The layouts use the crash-handling technique of GTI with a fan-in of 11, i.e, 11 application processes and one tool place per compute node. In order to have a similar application process placement for both the reference runs and the tool runs, reference runs use 11 application processes per node only. This choice impacts the slowdowns of some applications that exhibit slowdowns below 1, e.g., *142.dmilc* for the *P2PMatch* layout. In these cases the reference runtime with 11 processes per node is higher than for 12 processes per node. A lack of process to core pinning could explain this behavior that remains a target for future study.

Both the *P2PMatch* and the *CollectiveMatch* layout exhibit very similar slowdowns as in the fan-in 4 measurement in Section 6.3 (page 124). However, the increased fan-in increases the slowdown for applications such as *128.GAPgeofem* or *143.dleslie*. This impact becomes more visible in the slowdowns for the *TransitionSystem* layout in Figure C.3(c). Slowdowns for *121.pop2*, *128.GAPgeofem*, and *143.dleslie* rouhgly double compared to the results for fan-in 4. As a consequence, the measurements report no result for *128.GAPgeofem* at 2,096 processes as to avoid an excessive use of CPU quota. The *TransitionSystem* layout with fan-in 11 still detects and reports the deadlock for *126.lammps* and exhibits similar detection overheads as for fan-in 4.

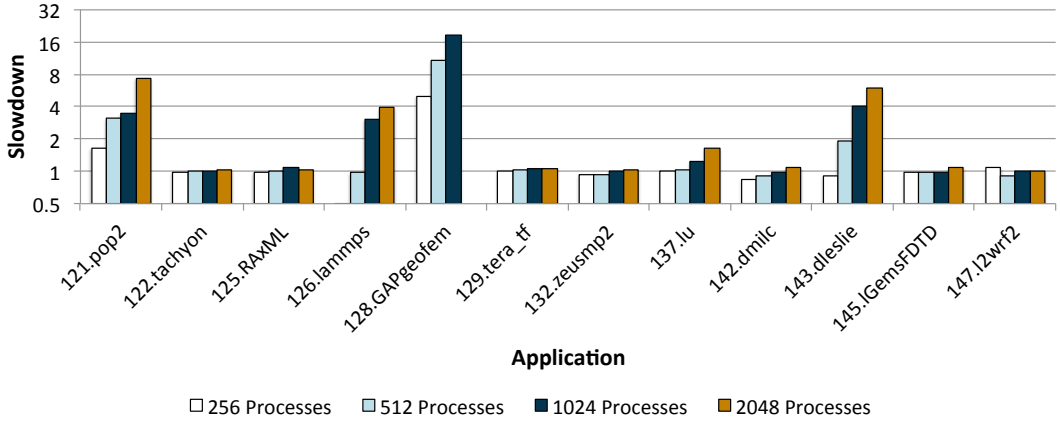
Overall, the measurements with the application crash-handling technique exhibit higher slowdowns that primarily result from the increased fan-in. The *P2PMatch* layout yields a slowdown above 2 for two applications and the *CollectiveMatch* layout for one application. Excluding the application *126.lammps* (with deadlock), the *TransitionSystem* layout yields a slowdown above 2 for three out of twelve applications. As for the results with fan-in 4, the actual use case—especially runtime of the application—determines whether the higher slowdowns remain acceptable for the use of the tool.



(a) For the *P2PMatch* module.



(b) For the *CollectiveMatch* module.



(c) For the *TransitionSystem* module.

Figure C.3: *Slowdowns for the SPEC MPI2007 applications on Sierra with MUST's application crash-handling scheme.*

Acknowledgments

First and foremost I want to thank my mentors and advisors Wolfgang Nagel, Matthias Müller, Bronis de Supinski, and Martin Schulz. You have served as my examples, teached and guided me, or supported me when needed. Bronis, I do not know how you can mange with all the responsibilities you have, but especially to you goes my gratitude for dropping me a line when I needed it most.

I want to thank the ASC Tri-Labs and the Los Alamos National Laboratory for their friendly support that enabled the development of the first prototype versions of GTI and MUST. Further, this work has been supported with funding from the European Community's Seventh Framework Programme project CRESTA. I am also deeply grateful to Prof. Nagel for the trust that he had in me during the early phase of this project. My deepest appreciation for your continued support of my ideas.

The experiments in this thesis used allocations on HPC systems at the research center Jülich and at the Lawrence Livermore National Laboratories. The ability to perform these measurements enabled an evaluation of the concepts that this thesis proposes, many thanks! Appologies go to the admins of both sites for repeatedly crashing their nodes, for exhausting their memory, and for using up all these cycles.

The Center for Information Services and High Performance Computing in Dresden was always a wonderful place to work at. Many thanks to all the colleagues at the center for the great time there (and hopefully there are many years to come). I want to especially thank Andreas Knüpfer for his advice and for proof-reading my thesis.

Finally, I want to thank my wife for her support and patience.